2001

Broadcast distributed shared memory

Philip Ragner Auld
College of William & Mary - Arts & Sciences

Broadcast Distributed Shared Memory

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Philip R. Auld

2001

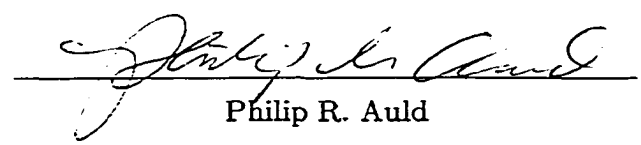
Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

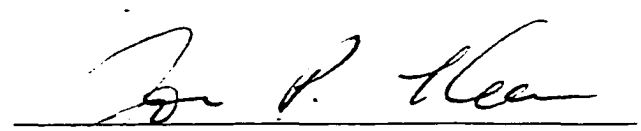
the requirements for the degree of

Doctor of Philosophy

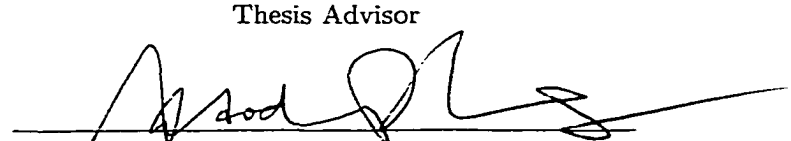


Philip R. Auld

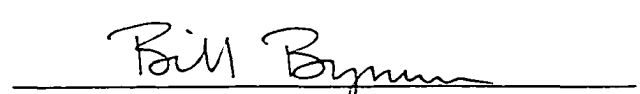
Approved, January 2001



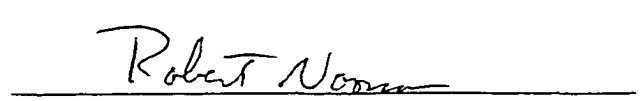
Phil Kearns
Thesis Advisor



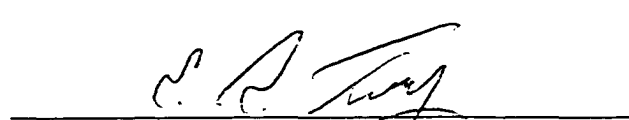
Xiaodong Zhang



Bill Bynum



Bob Noonan



Gene Tracy
Department of Physics

To my father. Dr. Louis E. Auld

Table of Contents

ACKNOWLEDGMENTS

I would like to thank my advisor, Phil Kearns, for his help, guidance and patience. Matt and Tracy deserve my thanks for encouraging me to take the first steps toward this accomplishment. I owe a great deal to my wife Catherine, love and eternal gratitude for all the encouragement, proof-reading and most of all patience. Thanks to Anna, Joel, and Felipe for many discussions and letting me bounce ideas around. Finally, thanks to my mother and father for everything.

List of Figures

ABSTRACT

Distributed shared memory (DSM) provides the illusion of shared memory processing to programs running on physically distributed systems. Many of these systems are connected by a broadcast medium network such as Ethernet. In this thesis, we develop a weakly coherent model for DSM that takes advantage of hardware-level broadcast. We define the broadcast DSM model (BDSM) to provide fine-grained sharing of user-defined locations. Additionally, since extremely weak DSM models are difficult to program, BDSM provides effective synchronization operations that allow it to function as a stronger memory. We show speedup results for a test suite of parallel programs and compare them to MPI versions.

To overcome the potential for message loss using broadcast on an Ethernet segment we have developed a reliable broadcast protocol, called Pipelined Broadcast Protocol (PBP). This protocol provides the illusion of a series of FIFO pipes among member process, on top of Ethernet broadcast operations. We discuss two versions of the PBP protocol and their implementations. Comparisons to TCP show the predicted benefits of using broadcast. PBP also shows strong throughput results, nearing the maximum of our 10Base-T hardware.

By combining weak DSM and hardware broadcast we developed a system that provides comparable performance to a common message-passing system, MPI. For our test programs that have all-to-all communication patterns, we actually see better performance than MPI. We show that using broadcast to perform DSM updates can be a viable alternative to message passing for parallel and distributed computation on a single Ethernet segment.

Broadcast Distributed Shared Memory

Chapter 1

Introduction

The use of a network of workstations (NOW) as a computational platform has increased in recent years as the price and performance of single processor machines and network hardware has improved. These clusters are being used in two important ways. The first is as a high performance compute engine. Performing parallel primarily numerical computations on a cluster can provide results approaching those of dedicated parallel machines at fractions of the cost[11, 16, 23, 25, 33, 42, 81]. The second major application uses the clustered machines for redundancy, to remove the single point of failure and bottleneck of single server systems[1, 48, 81]. These networks of workstations are used as reliable, intranet, distributed platforms, for example: network file system (NFS) servers, distributed databases, and distributed web servers.

The relatively small cost of a cluster of workstations compared to the cost of a high-performance computing platform, as well as the potentially higher accessibility provided by multiple semi-autonomous workstations, has lead to substantial work in the area of parallel computation on a local area network (LAN) of workstations[25, 33]. More people can use

the workstations as desk machines allowing the costs to be dispersed, making better use of the systems. In order to make them useful for parallel computation these systems need to exhibit performance close to that of the parallel machines they are replacing. The price and utilization benefits of the NOW model can overcome some of the performance gap between these systems and dedicated high performance platforms. This can be seen by the growing use of Beowulf[83, 84] systems. In fact, the cluster model is being applied to dedicated systems built specifically for such uses. While many of these systems use special high bandwidth switching hardware as an interconnect, a cluster of processing nodes on a switched Ethernet[65] segment is a common platform for parallel programming.

Many computing tasks, where performance is not as vital as reliability or availability, can benefit from the distributed network model as well. For example, a distributed database might not have the same performance requirements as a parallel numerical computation, but might be required to survive longer. An important aspect of a distributed platform can be fault-tolerance. A system of dispersed processors should be less prone to total failure than a single multiprocessor machine. Again, a database needs to survive a processor failure and potentially recover its state without restarting.

These platforms often require either a layer software to manage the distribution of the computation or changes to the program itself. The transition from a parallel program running on a multiprocessor to one running on a LAN is not an easy one. Many multiprocessors provide shared memory, allowing programs to be written in a manner similar to single-processor programs, using accesses to shared memory locations as the principal means of inter-process communication. A number of cache coherency protocols have been developed to ensure the correct execution of programs running on tightly coupled multipro-

cessors. Due to the often higher latency and possible inability to snoop, cache coherency protocols may not adapt well to a truly distributed system. Therefore, most distributed programs are written using the traditional message-passing model, treating each process as a node to which explicit messages are sent. This style of programming is not straightforward: processes must be coordinated, addresses established and connections made and maintained. Small changes in the computation algorithm can have major consequences for the programmer. The programmer spends a great deal of effort on the details of message-passing, taking time away from the actual algorithm that is the essence of the program. The apparent difficulty of programming parallel computations on distributed platforms has lead to the use of Distributed Shared Memory (DSM). Cheriton argues that shared memory programming is less difficult than message passing[32]. Many other authors take this as an assumption[5, 27, 40, 49, 67, 72, 87, 92]. DSM is the logical extension of the common, shared-memory, coherent-cache, multiprocessor paradigm used on many high performance machines. There has been much work in the area of DSMs in the past 10 years or so.

1.1 Distributed Shared Memory

On a single processor, a process has a well-defined relationship with the memory. This relationship is sometimes known as the register property. A single register or memory location can have one bit pattern in it at any one time. All of the bits get set in parallel, so a change is atomic. Therefore, a load, or read, of this register or location returns one and only one possible value–the value most recently written. Sequential programming on a system obeying the register principle is relatively easy. One knows what to expect. The

value of a read can be determined by looking at the most immediate write in program order, taking into account any compiler optimizations. This model extends to early parallel processors with shared memory. The hardware forces a serial order on writes and reads from different processes. Atomic synchronization operations can be used to force a certain global order. Threaded programming on a single processor obeys this model, with the addition of synchronization, although there is no truly concurrent execution. However, when the register property model is extended to a distributed system the notion of "most recent" becomes less well-defined. Clock skew, message-passing delay, and different processor speeds all contribute to the lack of a strict global notion of one event's happening before another. We no longer have a single hardware location to enforce a sequential order. It is not always possible to determine which is the most recent write.

The original proposals of DSM attempted to use the same model of memory as the single processor. Li and Hudak[62] present a system that mimics virtual memory, except that not only can pages be on secondary storage they can be on a different machine. Stumm and Zhou[87] covered a range of algorithms that provide DSM. Their paper describes several types of systems that implement a readers/writers protocol, using both write-update and write-invalidate. One such system used pages of shared memory and allowed any number of copies to be disseminated for reading by the page's owner or a central managing process. When a process gains write access, all of the readers must invalidate their copies. They would then have to request a new, updated copy on the following read fault. These systems were obvious extensions of virtual memory and cache coherency protocols. Another implementation was to send out an update of the new locations after a write (or series of writes). The idea was to keep down the size of messages on the network by not sending the

entire contents of a memory page. Here also, there was a need for a single process to acquire write access to the appropriate pages before sending out any updates. In this way, strict coherence is maintained. There is still only one global view of what is in the memory. Any one process may not see the entire contents, but what it does see will be the same as every other process' view of that portion of the shared data-space. Maintaining this strict view of memory is very useful from a programmer's point of view. Single processor programs can be distributed and run on a strict DSM with very little modification. However, network latency exacts a serious toll. Not only do data pages need to get from one processor to another, but write access requires a distributed mutual exclusion protocol. Whenever a process tries to write to a page for which it doesn't have write permission, globally-exclusive access must be established. Page size also plays a major role in the performance of such systems. While larger pages require less network communication, they also mean less concurrency as more processes can be competing for a given page.

As a result of the fact that DSM systems were many times slower than message passing parallel programs, researchers began to examine the underlying memory model for a way to increase performance[4, 28, 27, 60, 63, 87]. The following sections describe some of the models that have been developed in an effort to reduce the overhead of strictly consistent DSM, while allowing programming ease. Most of these models are designed for scientific computations and, therefore, strive to provide at least the illusion of a coherent memory.

1.1.1 Improving DSM performance

Early DSM systems enforced a global notion of memory coherence or consistency. This consistency, while making programming almost as straightforward as sequential processing,

was not without cost. Implementing a readers/writers protocol over the network greatly reduced concurrency as many processes had to continually wait for access to pages of memory. In response to this poor performance, several so-called *weak memories*, which relax the consistency constraints, were proposed. The consistency constraints of a memory model are the guarantees provided to the user about how the memory will behave. For example, a uniprocessor memory has consistency constraints that ensure a sequential order of memory accesses directly related to the program order of a process running on the memory. A read is guaranteed to return the last value written to a location, with *last* defined by program execution order. These weaker models reduce the consistency constraints on the memory, allowing executions to become incoherent in an attempt to increase performance. By incoherent we mean the memory can be in a state where two reads of the same location by different processes can yield different values. Or, in other words, processes can see different executions, something that cannot happen in strictly sequential processing. Again, some of these weak models are analogues of conventional coherence schemes for shared memory multiprocessors. Others are derived from the notion of causality inherent in distributed systems. Section 1.2 presents a summary of the consistency provided by some of the weak models. One of the problems with these weak memories is that programming is more complex than it is for the consistent models. Allowing the memory to become incoherent means different processes can have different views of the shared memory. Most of the more widely used models of weak DSM impose programming constraints that, if followed, allow the memory to appear consistent. In other words, the programming model is constrained to ensure the memory appears coherent to the user program. This allows the performance benefit of weakening the consistency to coexist with a viable, known programming model.

Most of the work on DSM has focused on page-based systems. Most of these systems extend the built-in memory paging facilities to include a protocol for distributed operation. Several of these systems, using weak memory semantics, have achieved acceptable performance increases over similar single process programs. In fact, at least one, Treadmarks[9], is now commercially available. This and similar systems have the advantage of having transparent memory accesses. However, these page-based systems have several drawbacks. First, sharing among processors is limited by the page size. Sharing units smaller than a page, which is often 4KB, can create unacceptable delays as a page containing more than one item is swapped among processors. Either care needs to be taken to lay out data on separate pages or extra protocol needs to be added to facilitate the sharing of a single page. Second, these systems tend to have very complex protocols for ensuring memory coherence. The use of multiple-writer protocols leads to a further increase in system complexity. Even though page-based DSM has been explored more thoroughly than update systems, we feel that for certain hardware and software situations update-based DSM, with its simpler protocol, can be a viable option.

The Munin[26] shared object system uses the page-based model to implement object level granularity using release consistency with multiple writers. This system uses multiple threads and object-level, or location-level, granularity. As it is page-based, it transmits whole pages using point-to-point communication as its method of propagating writes to other processes.

Several systems use a form of automatic update to provide weak DSM. The SHRIMP system[50] uses special hardware to improve the performance of Lazy Release Consistent (LRC)[55] DSM. Oguchi, Aida and Saito[68] present the design and prototype of a DSM

system that uses multicast on ATM-based WANs. The design of this system is based on the FIFO order of multicast messages. It uses point-to-point messages to regulate access permission to memory locations by way of a centralized semaphore server. The implementation of the ORCA[15. 89] programming language uses broadcast to disseminate updates to distributed objects. These updates are serialized through a central process to provide a global order.

1.2 Memory Coherence Models

In order to clarify discussion of memory coherence models. we present an overview of the basic models of coherence. For this overview we will begin by looking at the most strict model and move to weaker versions. comparing and contrasting them as needed to further understanding. There are two classifications of weak DSM models depending on the nature of reads and writes. Those in which read and writes are all the same type of access we label "generic read/write" models. Those in which some reads and writes are special accesses and behave differently we call "special access" models. The DSM model in our research is of the first type so we begin looking at the generic models. We then discuss some of the special models because the coherence in our model is similar to that provided by some of these.

1.2.1 Sequential Consistency

Modern parallel machines do not follow the atomic. serial model of memory coherence. That is, the steps of one access can overlap the steps of another. In fact, they can be executed out of program order. However, they are made to appear to happen in program order to all

processes. Lamport[59] defines sequential consistency (SC) as follows: "A multiprocessor is said to be sequentially consistent if the results are the same as if the operations were executed in some sequential order, and all of the operations from any one process are in the order specified by that process's program." The term *operation*, in our case, refers to any access of shared memory. For the examples used in this paper all memory locations are assumed to have initial values of 0. We use the notation "$z := 1$" for assignment, or a write, of the value 1 to location z. Similarly, "read(z) = 1" is a read of location z that returns the value 1.

P_0	P_1	P_2
$z := 1$;	while (x != 1) skip;	while (y != 2) skip;
$x := 1$;	read(z) = 1;	read(x) = 1;
	$z := 3$;	read(z) = 3;
	$y := 2$;	

Figure 1.1: Sequential Consistency Example Programs

As an example of sequential consistency consider the simple programs in figure 1.1. The locations x and y are effectively used as synchronization operations. The values read by P_2 are the only possible results. In sequential consistency, all processes have the same view of the order of events in the execution. There is one global view, which is a single interleaving of events, established by the execution history and shared by all processes. Sequential consistency is the model most used in modern computing platforms from the programmer's point of view. Memory accesses behave most like they do for a single processor. This allows programmers to use the memory model with which they are most comfortable.

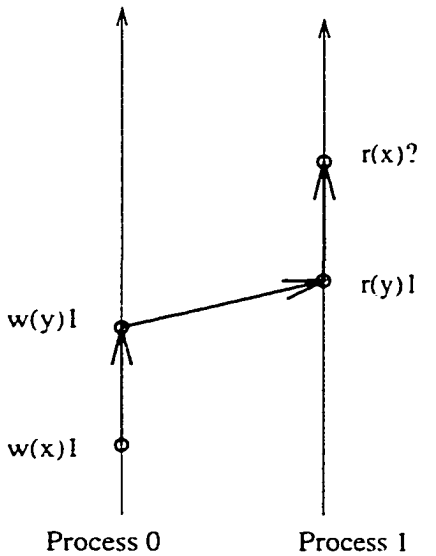


Figure 1.2: Simple Causal Relationship.

1.2.2 Causal Memory

While the SC model grew out of the multiprocessor domain, causal memory takes a different approach, drawing its power from the causal nature of the communication is a distributed computation. Lamport[57] formalized the notion of causality in concurrent computations in the "happens before" relation ($\rightarrow$) on system events. This relation was applied to DSM by Ahamad, Hutto and John[5]. The causal memory model is defined in terms of what value can be returned by a read operation. Essentially, a write is analogous to a send operation and a read is analogous to a receive. Here, however, a single write can have many reads return its value. When a write is received into a processor's view of memory, further reads to that location will see that value until it is overwritten.

Definition - A memory system is said to be causal if a read returns the most recent write

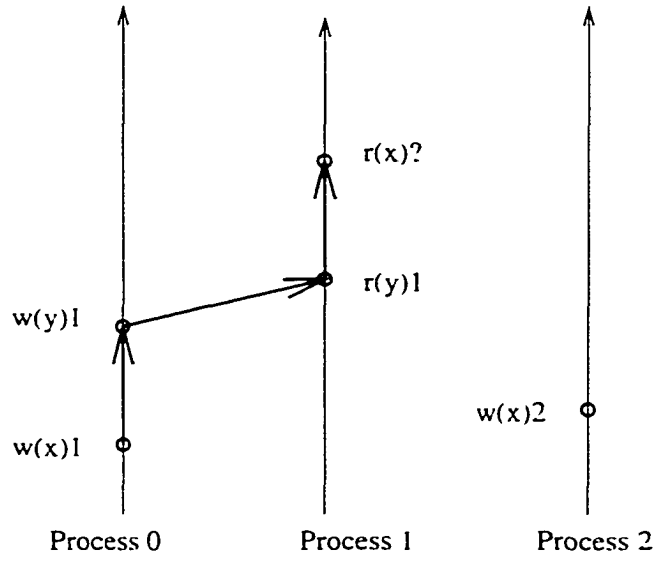


Figure 1.3: *Concurrent Writes.*

as defined by the relation $\rightsquigarrow_c$.

The relation $\rightsquigarrow_c$ is then the transitive and irreflexive closure of $\rightarrow_c$, where $\rightarrow_c$ is a translation of "happens before" as follows:

1. If operation o and o' are successive memory operations by the same process, then $o \rightarrow_c o'$.

2. If the read operation o_r returns the value of write operation o_w, then $o_w \rightarrow_c o_r$.

3. Additionally, if not $o \rightarrow_c o'$ and not $o' \rightarrow_c o$ then o and o' are said to be concurrent.

It is possible for the return value of a read to be an element of a set of possible values made up of the values of all of the writes that are concurrent to the read in question, or are concurrent to a causally preceding write.

In figure 1.2, the two parallel arrows represent the time-line of each process. We use "$w(y)1$" to mean a write to location y of the value 1 and "$r(y)1$" to mean a read returning 1. A "?" is used when a returned value is unspecified. It is easy to see the causal relationship between $w(x)1$ and $r(x)?$, shown with the bold arrows. The fact that the previous read event, $r(y)$, read the value 1, means that process 0 has executed past $w(x)1$. Therefore, that value must be available to be read, in the absence of a later write. In figure 1.3, there is no causal relationship between the writes to location x by processes 0 and 2. They are, therefore, concurrent writes. In process 1, $r(x)?$ now can return either 1 or 2.

The implementation of causal memory is documented, and a formal programming model is defined in which synchronization variables are used to force the causal relationship to follow a prescribed path[6, 51, 52]. That is, synchronization is used to order the reads and writes of a parallel computation in a strict fashion, thus ensuring each read operation returns the value it should under the sequential consistency model. Enough synchronization is used to ensure there are no data races in the program. In this way, a weaker form of memory is transformed into a sequentially consistent memory, if the programming model is followed. This result is proven by John and Ahamad[52]. The example used for SC (figure 1.1) will execute the same way on causal memory as it did on SC.

1.2.3 Processor Consistency and PRAM

Processor Consistency (PC) is a weakening of SC that orders events only based on the program order of the issuing process. PC was introduced by Gharachorloo et al[41]. As with SC, the operations by any one process must be seen by all processes in program order. However, unlike SC, the interleaving of operations from all processes need not be the same

as seen by all processes. In other words, there is no global ordering shared by all processes. Different processes can see different orders, within the bounds of program order.

Pipelined RAM, or PRAM, was first introduced by Lipton and Sandberg[63]. PRAM provides the same coherence model as PC. In this paper, we use the term PRAM for this model of coherence. PRAM is based on the idea of each processor having a copy of the shared data and a queue (pipeline) of incoming write updates. These queues receive write updates from other processors in the order issued by the writing processor. These updates can arrive in an arbitrarily-interleaved order with respect to write updates from other processors, as long as the updates of each process appear to all others in the program order of the writing process. These updates are then serviced, incorporated into the shared memory, in FIFO order. PRAM allows writes to be arbitrarily delayed. In fact, there is no guarantee the writes will ever take effect. The idea is to incorporate the network latency involved in propagating write updates into the memory model.

P_0	P_1	P_2
$z := 1$:	while (x != 1) skip:	while (y != 2) skip:
$x := 1$:	read(z) = 1:	read(x) = 0:
	$z := 3$:	read(z) = 3:
	$y := 2$:	

Figure 1.4: PRAM Example

For example, consider the processes shown in figure 1.4. It is possible for each process to have a distinct view of the shared data space. Process P_1 has clearly seen the value 1 in x. However, P_2 might not have, despite there being a "causal" link. The write to y which

allows P_2 to continue cannot happen until the write to x has been seen by P_1. Process P_2 still sees the value of 3 in z as it should because the writes in P_1 are seen by all other processes in the order issued. This example is one possible execution.

In the case of causal memory, synchronization is used to ensure the program sees the stronger SC consistency that it expects. Since most, if not all, parallel programs require some form of synchronization anyway, this is not excessively burdensome. With PRAM and weaker memories causality-based synchronization is no longer possible. It is still possible to have processes synchronize their executions, but it may not appear that way to other processes. A slight modification to the original example shows this. In figure 1.5 the processes are synchronized to execute in sequence. However, P_2 does not see the results computed by P_0 before the synchronization.

P_0	P_1	P_2
$z := 1$;	while (x != 1) skip;	while (y != 2) skip;
$x := 1$;	read(z) = 1;	read(x) = 0;
	$y := 2$;	read(z) = 0;

Figure 1.5: PRAM fails logical synchronization

1.2.4 Slow Memory

Slow memory presents a very relaxed view of memory. While most of the previous memories preserve program order as an important aspect of the model, allowing some synchronization techniques to function as expected, slow memory doesn't enforce program order on the memory system. Hutto and Ahamad define slow memory as a location-relative weakening

of memory consistency [49]. They include it as an example of a multiversioning memory, making it suitable for application to distributed objects. Slow memory can be defined in terms of how a read works. A read on a slow memory location must return some value previously written to that location. Further, once a value has been read, no earlier write to that location by the process whose value is returned can be read. Writes by a process are always seen by that process immediately. This makes for a form of memory much weaker than all of the previous examples.

P_0	P_1	P_2
$z := 1$;	while (x != 1) skip;	while (y != 2) skip;
$x := 1$;	read(z) = 0;	read(x) = 0;
	$z := 3$;	read(z) = 1;
	$y := 2$;	

Figure 1.6: Slow memory example

Figure 1.6 shows a possible execution of the example programs on slow memory. We see that not only does P_2 not see the write to x, as in the PRAM example, it might not see the second write to z. Under PRAM, P_2 must see the value 3 in z due to the program order of P_1. With slow memory this is not the case. Synchronization is also a problem for slow memory. The example fails logical synchronization. Process P_2 does not see the results of P_1's computation prior to synchronization.

An interesting aspect of slow memory is illustrated by the solution to the dictionary problem [38]. The problem is to implement a simple associative table with *insert*, *delete* and *lookup* operations. *Lookup* should return all values inserted but not deleted. What

makes the problem difficult is the requirement to satisfy the following conditions:

1. The view must be consistent. An item is in a process's view if and only if it has been inserted and not yet deleted.

2. The system must be space-efficient, using bounded storage.

3. The system must be fault-tolerant. Functioning processes must continue, despite other processes or communications failing.

4. All views must eventually converge and become consistent if there are no further inserts or deletes.

Ahamad and John propose a solution to this problem using slow memory[49]. This solution demonstrates part of the power of slow memory. We feel that slow memory is an interesting alternative to the earlier mentioned weak memories.

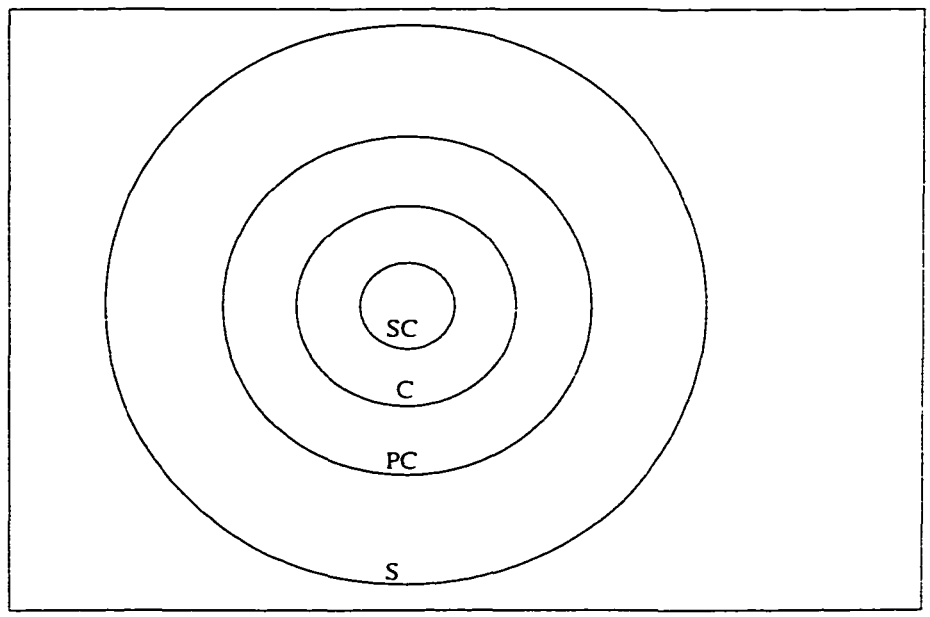
1.2.5 Relating generic weak memories

The weakness of a memory model can be seen as a space of allowable executions. The larger the space the more concurrency is allowed, at the expense of tighter event ordering. The generic memories presented above are related by

$$SC \subset Causal \subset PRAM \subset Slow.$$

That is, all SC executions are causal, but not all causal executions are SC[49, 73]. All legal causal executions are also PRAM, but there are legal PRAM executions that violate causality. The same relation holds between PRAM and Slow. Our running example shows

these relationships. For example, figure 1.1 is a legal Slow memory execution, while figure 1.6 is not a legal SC execution. This relationship is shown pictorially in figure 1.7.



SC = Sequential consistency
C = Causal
PC = Processor Consistency (PRAM)
S = Slow memory

Figure 1.7: Coherence models as execution spaces

1.2.6 Special Access Weak Models

Special access weak models are those that differentiate between memory accesses. The earlier models all assume reads and writes are the same. That is, one write is the same as any other. Special access models have been developed that make a distinction between types of memory accesses. Some are general and behave as the previous models. Some have additional impact upon execution. For example, a write access may be part of a synchronization operation and have tighter ordering restrictions. These models are often originally hardware based, like Weak Ordering[3] and Release Consistency[41]. Lazy Release

Consistency[55] is a modification of Release Consistency that is designed for a software DSM system. In this section we look at the basic operation of these models for completeness.

1.2.6.1 Weak Ordering

Like PC, Weak Ordering (WO) was derived from hardware-based multiprocessors. In this case, there are constraints put on the software to be run on the WO system. These constraints, if followed, allow the memory to appear sequentially consistent without paying the full price of a truly SC system. Memory accesses are broken down into two types: accesses to synchronization variables, and accesses to normal variables. The WO model can be summed up as follows.

Definition : A system is said to be weakly ordered if:

1. Access to global synchronization variables is strongly ordered (synchronization obeys SC).
2. No access to a synchronization variable is allowed until all previous shared memory accesses are globally performed.
3. No access to global variables, synchronization or normal, is allowed until all previous synchronization accesses are globally performed.

Another way to see this is from the software point of view. A memory system is weakly ordered if software obeying a "synchronization model" appears to be running on sequentially consistent memory. Adve and Hill[3] present a number of synchronization models for weakly ordered systems. The primary model is called Data-Race-Free. In this model, the notion of causality, as codified by Lamport[58], is captured in the context of memory accesses. A

data race is defined as any two operations not strictly ordered by causality, that is to say, concurrent operations, where at most one is a read. Intuitively, this is what one expects. Concurrent read accesses are acceptable, concurrent writes are not.

1.2.6.2 Release Consistency

Release Consistency (RC), is similar to weak ordering. RC is a slightly more formalized memory model often applied to software DSM systems. RC is based on the observation that the structure of parallel programs allows the memory to become incoherent for certain portions of the execution, returning to a coherent state at synchronization points, e.g. the barrier between iterations, without sacrificing correctness.

Originally presented as part of the DASH multiprocessor[41], release consistency gets its name from the synchronization operations *acquire* and *release*. These synchronizing (or *special*) operations are analogous to lock and unlock in a standard synchronization model. RC is defined as follows.

Definition - A memory is release consistent if:

1. Before an *ordinary* load or store operation is allowed to be performed, all previous *acquire* operations must be performed.
2. Before a *release* is allowed to be performed, all previous load and store operations must be performed.
3. *Special* accesses obey processor consistency with respect to each other.

Programmers on a release consistent system are required to label memory accesses as acquire, release or ordinary. If the program is properly labeled, then the memory will appear

sequentially consistent. and a correct program will execute correctly. *Properly labeled* is defined as having *enough* special accesses labeled either acquire or release[41]. *Enough* means that for any two memory accesses on different processors. if at least one is labeled ordinary and one of the two comes before the other in a correct execution. then there is at least one release on one processor and an acquire on the other. depending on which needs to be before the other[40. 43]. The idea behind RC is that time is only spent ensuring sequential ordering among those accesses labeled as *special.* Because of the semantics of critical sections. the writes can happen in any order as long as they are all seen by other processors before the release is completed. This is also used to reduce communication. The writes can be buffered until the release operation and then sent out all at once. Carter. Bennett and Zwaenepoel[28] provide a very detailed look at how these techniques can be used to further increase the efficiency of the RC model. In fact. results close to those of hand-coded message passing can be achieved using modern implementations on RC.

A variant of RC is Lazy Release Consistency (LRC)[55]. While RC sends invalidation messages to all other processes on a release. LRC exploits the causal relationship to send these invalidations only to the next process that acquires the page in question. This significantly reduces the amount of message-passing overhead. This software extension is the basis of the Treadmarks system[9]

1.3 Goals of This Research

Extremely weak memories, such as PRAM and Slow, have several benefits that make them attractive. First, there is no causal link between writes from different processes. This

means that there is no global ordering, and write updates can be applied as they are received. Therefore, such weak memories should be efficient. Second, they are inherently update-based. When a process issues a write it is effective immediately on the local copy of memory. Then the new value for that location is sent as an update to the other processes. Thus we have a system of fine-grained memory accesses. The locations can be of any size. Third, having multiple copies of the shared memory space allows for a higher degree of availability than page-based systems which may have only one valid copy of a given page.

The weakness that allows non-causal memories to perform well is also a hindrance to meaningful programming on them. In the case of causal memory, synchronization is used to make the memory model appear stronger than it is to allow effective programming. With PRAM and Slow there is no causality enforced on the order of writes so it is not possible, using memory locations, to have enough synchronization to make the memory appear sequentially consistent. In order to make a similarly weak model usable, some method of effective synchronization is required.

Update-based DSM systems can generate a number of communication messages. Each and every write needs to be passed, usually as a message in an underlying system, to each other process. However, many cluster computing environments where DSM might be used are local area networks. Further, many of these networks are Ethernet based. This means there is the possibility of using hardware-level broadcast to increase the effective bandwidth of sending updates. We are interested in determining if reliable broadcast can be used to increase the efficiency of update-based DSM systems.

1.3.1 Using Broadcast For Communication

Sharing data among processors on a local area network requires communication. Often, this communication involves sending messages containing the same information to a number of other processors. This is especially true for systems that replicate data across the entire set of participating systems. Distributed databases and other systems that provide high availability, and shared memory or object systems that work on the update model are examples. This kind of communication is also common in many parallel numerical computations where each process needs values computed by the other processes to continue. A common communication pattern in many such programs involves each process both sending and receiving data from each other process, so-called all-to-all communication. Since many of these systems are on a broadcast medium network such as Ethernet, we feel that by using hardware broadcast we can perform this data movement more efficiently than with point-to-point messaging. However, using UDP/IP broadcast on an Ethernet segment can be subject to faults of omission. Single packets may be lost due to corruption or buffer overflow, at the receiving or sending process. Most computations will not tolerate this loss of data. Some mechanism is required to ensure delivery of each packet. One solution for message-passing systems for parallel programming is to use the connection-oriented TCP protocol. However, this means the broadcast medium is not being used. An n process system requires n^2 point-to-point TCP streams to support all-to-all communication. Another solution is to build an acknowledgment scheme into each program as needed. A third approach is to develop middle-ware for reliably sending broadcast messages.

This aspect of our research is based on this third approach. We have developed a reliabil-

ity protocol for UDP broadcast packets on a single Ethernet segment. We are interested in a low-level protocol analogous to the (usually unimplemented) reliable datagram protocol. RDP[70]. We feel there is a need for a protocol that does not have the overhead of a reliable, atomic, or totally-ordered, broadcast system designed for a general range of networks. The system we explore is designed for a single network segment so there is no need to handle routing or to use a software emulation of broadcast. Also, because of the small scale of the network setup we don't need to address more complex problems like assuring virtual synchrony and recovering from network partitioning. Our protocol simply needs to ensure delivery of distinct datagram packets. In order to allow efficient, correct parallel computing, we also want the protocol to ensure that messages sent by any process are seen by others in the order sent. We are not interested in globally ordering messages, either totally or causally. That is, the order we provide is based solely on the sequence of messages sent by each process. There is no interdependence among messages sent by different processes. Globally ordering messages goes beyond the scope of a simple UDP level protocol. One can think of a collection of FIFO pipelines connecting each process to each other. The protocol we have developed is therefore called Pipelined Broadcast Protocol (PBP) to stress its kinship to UDP and RDP.

1.3.1.1 Strong Reliability

Much of the work in the area of reliable broadcasts uses a strict definition of reliable. There have been a number of reliable broadcast protocols presented in the literature[30, 22, 2, 20, 35, 66, 18]. However, they are primarily concerned with a stronger definition of reliability. Most of these systems take reliability to mean an atomic broadcast, despite process failures.

One of the motivations for these systems is to provide virtual synchrony among distributed processes. Each broadcast message is guaranteed to be seen, accurately and in order, by all non-faulty processes or by no non-faulty process. This is especially useful to distributed database systems, but is more strict, and time consuming, than is required for many parallel programs. Most of the protocols also work for general network configurations and often incur greater overhead than our system because they provide greater service. We briefly discuss some of the major reliable broadcast systems and point out some of the ways they are different from PBP below.

Starting with ISIS[18], researchers have looked at protocols to achieve atomic broadcasts in the presence of process or network failures including lost messages. The ISIS system provides for causal or total order and ensures virtual synchrony among the processes. The notion of virtual synchrony is essentially a form of agreement. Each process will see every message sent even if a sender fails after sending messages to some process but before sending to the others, or none will see such a message. The ISIS system is a more general system that provides more powerful service guarantees. It doesn't use hardware broadcast as it is designed to function on more diverse networks that may not have true broadcast capabilities.

The ORCA[15, 89] Reliable broadcast system for shared objects uses a serializing method. While it does use hardware broadcast, it only uses it to send messages from the serializing process. It takes at least two messages for each broadcast because each sending process must send its message to the serializing process, which then broadcasts it to the group.

Chang and Maxemchuck[30] developed a totally ordered protocol explicitly and exclusively for broadcast networks. Their system is somewhat similar to ours in that it was developed for the same specific network configuration. However, much of the complexity

of their system comes from the requirement to provide total order among messages. The system uses a rotating token to determine which process will acknowledge each broadcast message. The token-holding site, in effect, becomes a serializing influence. The other processes will deliver messages in the same order they are acknowledged by the token site. This means all processes deliver messages in the same total order.

1.3.1.2 Weak Reliability

Some work has been done to take advantage of broadcasting messages without providing all of the guarantees of strong reliability, while still ensuring delivery of all messages. The PSync[71] system uses piggy-backed acknowledgments and causal knowledge to determine message order and delivery. The PCODE[22] system is most similar to our approach. It uses hardware broadcast and doesn't provide a global, total order. However, it is demand driven and, in a sense, is synchronous. A receiving process makes requests to be sent messages. The Transis[8] system provides different levels of order, using multicast groups. The system provides for causal to total order of message delivery. Totem[66] and RMP[94] are similar systems that use a rotating token. Totem uses it to determine which process may send. RMP uses the token to pass information about delivered messages to allow buffer space to be cleared. They both provide causal or totally ordered message delivery within a multicast group. Both of these are systems designed for general network topologies. Therefore, they must resort to point-to-point messages on non-broadcast media.

1.4 Organization

In the following chapters we will present the work we have done to design and implement a system to provide computing cluster users with a broadcast-based, fine-grained DSM system. During our work it became clear that the communication layer is interesting and possibly useful in its own right. This led to the separation of PBP from the DSM system, allowing the reliable FIFO broadcast to be used for other purposes. Chapter 2 presents the theoretical model for BDSM. We present the coherence protocol and a programming interface. We then prove our system model provides the same consistency as PRAM. Before presenting the actual implementation of BDSM we explore the communication layer that was developed to support reliable broadcasts. In chapter 3, we present two versions of PBP. We present the protocol and provide a formalism that shows the system works as required. We have performed a series of networking tests with PBP to help gauge its performance relative to common network protocols, TCP and UDP. These results are presented in chapter 4. Chapter 5 presents a discussion of the implementation of BDSM on top of PBP. We then prove that the implementation preserves the theoretical requirements of BDSM presented in chapter 2. To test the performance of BDSM we have developed a test suite of parallel computations. We compare their execution times to those of similar programs using MPI in chapter 6. In chapter 7 we look at two extensions to our BDSM system designed to address issues of scalability and fault-tolerance. We draw some conclusions based on our work in chapter 8. To illustrate the programming usage of BDSM we include BDSM and MPI versions of one of the test programs in appendix A.

Chapter 2

Broadcast Distributed Shared Memory

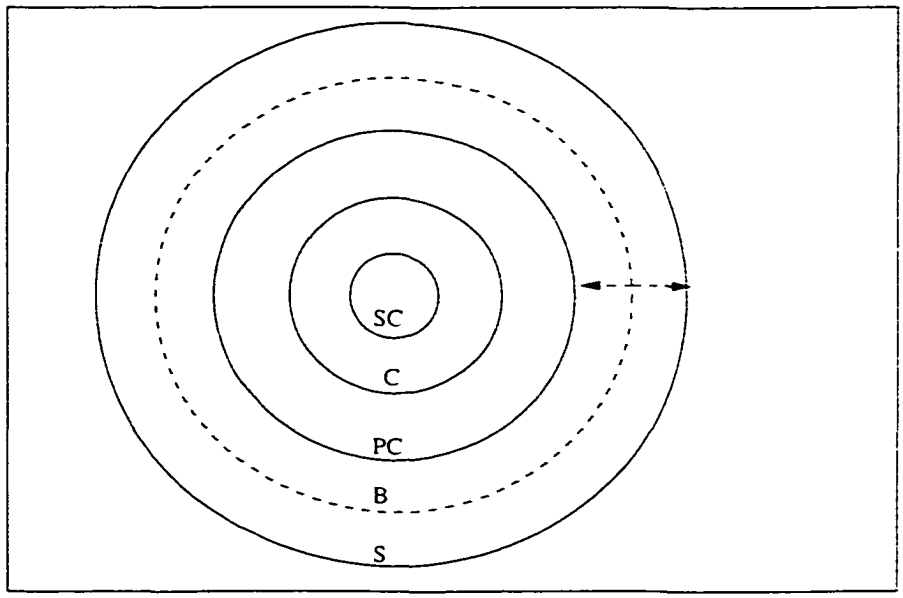
Distributed shared memory (DSM) is an interprocess communication method primarily for parallel computation. It strives to provide the same programming model for distributed memory machines as that found on many shared memory parallel systems. In this chapter we present the Broadcast DSM (BDSM) model we have developed. It is a weakly coherent model that uses broadcast communication to disseminate updates. It's weak enough to be efficiently implemented, but has strong enough synchronization that it can be used for meaningful programs. We show that the model can be made to appear sequentially consistent to programs that obey a certain programming paradigm. After presenting the protocol we discuss the interface to our prototype system.

2.1 BDSM Overview

The previous chapter presented some of the goals of our research. We are interested in developing a weak DSM that takes advantage of the efficiency of non-casual message passing. We feel there is potential for an update-based system that allows fine-grained access. We would like to utilize the broadcast capabilities of the underlying communication layer and hardware. To overcome the weakness of the memory model such a system should provide for effective synchronization. To achieve these goals we have developed a weak model we call Broadcast DSM (BDSM). This chapter presents the theoretical model and discusses its benefits. We show how the BDSM coherence model meets our stated requirements for a weak memory that is still programmable. Our system provides a model that is between PRAM and slow memory in coherence. This model has effective synchronization to make it appear sequentially consistent when needed.

A number of systems have been developed that use the page level of granularity[9, 27, 55, 62, 67, 72, 82, 87, 92]. This page-sized sharing can lead to thrashing. Our system uses a smaller granularity as defined by the programmer. It provides a fully-replicated shared memory that is modified by updates. These updates are sent using hardware broadcast to reduce the number of messages and reduce the cost of updating multiple copies. For performance reasons we do not enforce a strict coherence model. BDSM provides a form of PRAM consistency, but it also allows functional synchronization. This makes for a straightforward programming model that can be used for parallel numerical computations as well as fault-tolerant distributed applications.

Programs using the BDSM model define and join shared memory segments. Each seg-



SC = Sequential consistency
C = Causal
PC = Processor Consistency (PRAM)
S = Slow memory
B = BDSM

Figure 2.1: Coherence models as execution spaces, with BDSM

ment is made up of some number of identically sized locations. The updates to any given segment are in strict program order. Updates to different segments are not so ordered. All writes are immediately visible to the writing process. In a program with only one declared DSM segment, BDSM provides coherence identical to that of PRAM. For multiple segments writes are ordered by process and segment due to independent, per-segment buffers. Therefore, it is possible for two writes by one process to be applied by other processes in different orders. In this way, BDSM with multiple segments is weaker than the PRAM model. It can be though of as a hybrid of PRAM, which is processor relative weakening, and Slow Memory, which is location relative. The extreme case would be a program where each location was on a different segment. This would make the DSM look like Slow Memory to the

user program. Figure 2.1 shows the relation of the BDSM system to the other generic weak memories. Both of these extremely weak memories are considered to be too weak to program effectively. They can't be synchronized. The problem is that, by using shared memory locations for synchronization, there is not enough control to enforce a needed order. The memory cannot be made to appear suitably strong to execute meaningful programs. BDSM overcomes this problem by using the message-passing layer directly for synchronization.

Assume x and y are in different segments.

P_0	P_1	P_2
$x := 1$:	$y := 2$:	dsm_barrier(0):
$y := 1$:	dsm_barrier(0):	read(x) = 3:
$x := 3$:		read(y) = 1 or 2:
dsm_barrier(0):		

Figure 2.2: Coherence of different segments

By using broadcast-based synchronization we ensure consistency after synchronization operations. In this respect, our memory model is somewhat similar to release consistency. However, we do not guarantee that each process sees the same view of memory, just that all updates have been applied. Updates by different processes to the same location are not ordered. Therefore, even after a barrier, it is possible for two processes to have different views of memory. A program that allows unsynchronized access to the same locations by different processes may not have consistent views of memory across its subprocesses. As with other forms of weak memory, if this inconsistency cannot be tolerated by the application then more synchronization should be used. The fact that different segments are buffered and then updated independently will not effect the power of the synchronization operations.

When a process crosses a barrier it is confident that all updates from other processes issued before those processes reached the barrier have been applied. Since no location can be in more than one segment, the values in the various locations will be the last written in each case (barring data races).

P_0	P_1	P_2
$z := 1$;	while (x != 1) skip;	while (y != 2) skip;
$x := 1$;	read(z) = 1;	read(x) = 0;
	$y := 2$;	read(z) = 0;

Figure 2.3: Coherence provided is not causal

Figure 2.2 shows an example of three processes using two segments. We assume locations x and y are in different segments. Process P_0 makes two writes to x. Only the last one is guaranteed to be seen after the barrier, as shown in P_2. It would be impossible for P_2 to read the value 1 from location x. Additionally, there is a data-race involving location y. Therefore, its value is not determined. Reading location y after the barrier could return either value written. In order to remove the data-race, another synchronization operation would be needed after one write to y and before the other.

P_0	P_1	P_2
$z := 1$;	dsm_barrier(0);	dsm_barrier(0);
dsm_barrier(0);	read(z) = 1;	dsm_barrier(1);
dsm_barrier(1);	$z := 3$;	read(z) = 3;
	dsm_barrier(1);	

Figure 2.4: Coherence ensured using synchronization

It is important to note that the coherence model provided by BDSM is not equivalent to causal ordering. It is possible to have a write event by one process cause another process to issue a write which is then seen by a third before the initial write is seen (as shown in figure 2.3). Assume x, y and z are initially zero. Process P_2 can see the write to y without seeing the writes to x and z without violating FIFO update ordering. However, there is clearly a causal link between the write to z and that to y. Synchronization can be used to ensure causal ordering if required. Figure 2.4 shows the same basic model using barriers to enforce the required causal order. The *while* loops, and hence the writes to x and y, become unnecessary because the barriers serve to indicate a given write is completed. Note that the causal order is preserved. The write of 1 to z happens before the read in P_1. Since the assignment to z occurs before the barrier it must be visible to P_1 after it. Similarly, P_2 is guaranteed to see the value 3 in location z after crossing both barriers.

2.2 BDSM Memory Model

Processes access a shared memory space consisting of segments of equal sized locations. A write to a location is sent as an update to each other process in the system. Writes to any segment by a process are applied by all other processes in the order issued. The read of a location is a strictly local operation based on the latest value seen by the reading process. We define *seen* to mean a BDSM system message, an update or a synchronization message, has been delivered to and handled by the BDSM system at a given processor. For write updates this means the update has been applied to the local copy of memory. The meaning for synchronization operations depends upon the operation. A BDSM system message that

has not yet been handled has not been seen. An updated value can be seen without its having been actually read by the user process.

2.2.1 Definition of BDSM coherence

Define five operations: read. write. and three synchronization operations. The synchronization operations are barrier. lock_acquire and lock_release. The read and write operations provide access to individual memory locations. The synchronization operations do not have effect on memory other than to provide ordering. The lock_release operation only counts as a synchronization operation if there is a following acquire. That is. the synchronization is only meaningful if another process tries to acquire the lock. The relationships among these operations defines BDSM coherence:

1. Writes by any process to any segment are seen by all processes in the order specified by the program order of the writing process. Local writes are immediately visible to the writing process.

2. Synchronization operations are seen by all processes in the program order of the issuing process.

3. All writes issued by a process before issuing a synchronization operation are seen by all processes before the synchronization operation completes and all writes issued after are seen after.

Definition 2.1 *BDSM coherence: A DSM system that provides the required operations and preserves the above relationships is said to be* BDSM coherent.

In BDSM, a process that crosses a synchronization operation (passes a barrier, or acquires a lock) has seen all writes by other processes before that operation. In the case of a barrier, since it applies to all processes, all processes have seen all writes before the barrier. Locks behave differently. Acquiring a lock is a single process operation. All writes by that process will be seen by all other processes before the acquire completes. And, all writes by other processes will be seen by the acquiring process by the time it completes its acquire. There is no assurance that other processes see all writes by each other when the lock is acquired. Additionally, releasing a lock is a strictly local activity, unless another process has requested the lock since it was acquired. So, unless this is the case, releasing a lock does not order BDSM events.

2.2.2 Formalism and Definitions

In this section we present a formalism to model DSM behavior. The model consists of a collection of events and some useful definitions and relations. The ordering constraints on these events are supplied by the definition of the memory being modeled.

2.2.2.1 Events

Events are generically defined as $op_{i,n}^{a,b,c}$. The subscripts i and n are the same for all operations. The first, i, is the process in whose history this event occurs. This operation is local to process i. The second, n, is the time-stamp of the event in process i. Events in a given process are partially ordered by this time-stamp. A logical time-stamp is sufficient. Locally initiated events are totally ordered. External events that appear locally (updates and lock operations by other processes) are not so explicitly ordered – the time-stamp only reflects

when the event registered at this process.

The superscripts $a.b$ and c are used for optional information that is dependent on the value of op. They may be left off if they are irrelevant.

- Basic events:

  1. $r^x_{i.n}$ = read of location x by process i at local time n.
  2. $w^x_{i.n}$ = write to location x by process i at local time n. The actual value is immaterial.
  3. $u^{x.j.m}_{i.n}$ = update at process i at local time n produced by $w^x_{j.m}$.

- Synchronization operations:

  1. $b^k_{i.n}$ = barrier event k in process i at local time n. There must be corresponding $b^k_{j.m} \forall j$.
  2. $a^{k.i.n}_{j.m}$ = acquire event by i of lock k at i's local time n. If $i = j$ then $m = n$. if $i \neq j$ then this event represents j giving i permission to acquire the lock. If i acquires lock k ($a^{k.i.n}_{i.n}$) then there must be corresponding $a^{k.i.n}_{j.m}. \forall j \neq i$.
  3. $v^{k.i.n}_{i.m}$ = release event by i of lock k that was acquired at time n at i's local time m. These are local events. The importance is the next acquire. not the release itself. If another process is waiting for the lock then v is followed by the appropriate a operation.

2.2.2.2 Definitions

- Define $\prec$ as "ordered before". If a is ordered before b in a history then $a \prec b$. Note, this is not globally transitive. It is only transitive within a given process' sub-history.

as noted by the monotonically increasing subscript.

- Define concurrent, "$\|$", as $\neg(a \prec b) \wedge \neg(b \prec a)$.
- Define $\mathcal{E}$ to be the set of all events in the system.
- Define $\mathcal{H}$ to be the history of all events in the system. The history $\mathcal{H}$ consists of events in $\mathcal{E}$, partially ordered by $\prec$.
- Define h_i to be the sub-history of events as seen by process i. $h_i =$ set of all $op_{k.n}^{a.b.c}$ with $k = i$.
- Define $s(x)$ to be the segment on which location x resides.
- Define $c(op_i)$ to be the set of events concurrent to op_i.
- Define $last(r_{i.n}^x)$ to be the singleton set consisting of the most immediately preceding write or update event to location x in h_i.

  $last(r_{i.n}^x) = \{op_{i.m}^{a.b.c} : (op = w \vee op = u), a = x.m = max(k < n)\}$ Each process is assumed to issue $w_{i.0}^x \forall x$ to represent an initial base case.
- Define $val(r_{i.n}^x)$ to be the set of w or u events the written value of which r can return.

  $val(r_{i.n}^x) = last(r_{i.n}^x) \cup \{op_{j.m}^{a.b.c} \in c(r_{i.n}^x) : a = x \wedge op = w\}$.

2.2.2.3 BDSM Definition

BDSM can now be defined by a set of axioms that involve events. Most of them provide ordering constraints.

Axiom 2.1 *given* $op_{i.n}^{a.b.c}$ *and* $op_{i.m}^{d.e.f}$, $op \neq u \wedge n < m \Rightarrow op_{i.n}^{a.b.c} \prec op_{i.m}^{d.e.f}$.

Locally, all non-update events are in program order.

Axiom 2.2 $w^x_{i,n} \Rightarrow u^{x,i,n}_{j,m} \forall j \neq i$ *and* $w^x_{i,n} \prec u^{x,i,n}_{j,m} \forall j \neq i$.

Write leads to updates, and a write comes before its updates.

Axiom 2.3 $w^u_{i,n} \prec w^v_{i,m} \wedge s(u) = s(v) \Rightarrow u^{u,i,n}_{j,k} \prec u^{v,i,m}_{j,l} \forall j \neq i$.

Updates for writes to the same segment by the same process are seen in the order written.

Axiom 2.4 $b^k_{i,n} \Rightarrow b^k_{j,m} \forall j \neq i$.

Barriers are in all processes.

Axiom 2.5 $b^k_{i,n} \prec b^l_{i,m} \Rightarrow b^k_{j,o} \prec b^l_{j,p} \forall j \neq i$.

Barriers are totally ordered, and all processes see the same order.

Axiom 2.6 $w^x_{i,n} \prec b^k_{i,m} \Rightarrow u^{x,i,n}_{j,l} \prec b^k_{j,o} \forall j \neq i$..

Updates for writes before a barrier are seen by all processes before the barrier.

Axiom 2.7 $b^k_{i,m} \prec w^x_{i,n} \Rightarrow b^k_{j,o} \prec u^{x,i,n}_{j,l} \forall j \neq i$..

Updates for writes after barrier are seen by all processes after the barrier.

Axiom 2.8 $a^{k,i,n}_{i,n} \Rightarrow a^{k,i,n}_{j,m} \forall j \neq i$.

Lock acquires are seen by all other processes.

Axiom 2.9 $a^{k,i,n}_{i,n} \Rightarrow v^{k,i,n}_{i,m} \wedge a^{k,i,n}_{i,n} \prec v^{k,i,n}_{i,m}$

There must be a release for each lock acquired.

Axiom 2.10 $a_{i.n}^{k.i.n} \prec a_{i.l}^{k.j.m} \Rightarrow a_{j.o}^{k.i.n} \prec a_{j.m}^{k.j.m} \wedge v_{i.p}^{k.i.n} \prec a_{i.l}^{k.j.m}$

Lock acquires are ordered. and a lock-holder's release comes before the next acquire.

Axiom 2.11 $w_{j.n}^{x} \prec a_{j.l}^{k.i.m} \Rightarrow u_{i.p}^{x.j.n} \prec a_{i.m}^{k.i.m}$

Earlier updates by other processes must be seen before acquiring a lock.

2.2.3 BDSM coherence can be at least as strong as SC

Programming on weak DSM systems is usually done by making the memory appear stronger to a running program. This is done by using synchronization operations. A program running on what appears to be a sequentially consistent memory will behave as the programmer expects. BDSM can be made to appear at least as coherent as SC. To show that this is so we use the above formalism.

Definition 2.2 *data-race-free in the context of BDSM:*

1) Between any writes by different processes there is a global synchronization operation.

2) Between any writes by a single process to locations in different segments there is a global synchronization operation.

A global synchronization operation is one that effects all processes. In the BDSM case. there is only one type of global synchronization. a barrier.

Definition 2.3 *A global synchronization event e is said to be* between *two other events u. v if* $u \prec e \prec v$ *or* $v \prec e \prec u$.

This condition will be true if :

1. $u \prec e \prec v$ or $v \prec e \prec u$ in the program order of one process.

2. u and v are in different processes and e is a barrier then $u \prec e$ in one process and $e' \prec v$ in another. where e and e' are matching barrier events.

To capture the notion of a write being seen by all the processes in the system we use the term "globally precedes". If one write globally precedes another then that order is seen by all processes. That is. if a write and its associated updates come before another write and all of its associated updates then the first write globally precedes the second. We will use the $\ll$ operation to denote this. If all writes are globally ordered then the writes in a system are totally ordered. Each process sees the same order of write events. Writes in different processes can only be ordered by this relationship. If they are not so ordered they are concurrent.

Definition 2.4 *Globally precedes:* $w^x_{i,n} \ll w^y_{j,m}$ *if and only if* $u^{x,i,n}_{k,l} \prec u^{y,j,m}_{k,o}, \forall k \neq i,j \wedge u^{x,i,n}_{j,l} \prec w^y_{j,m}, i \neq j \wedge w^x_{i,n} \prec w^y_{j,m}, i = j$.

To prove that BDSM can appear as a sequentially consistent memory we will show that it can be made to provide a total order of all writes. This is stronger than SC. but is clearly sufficient to ensure at least sequential consistency. We first show what is required for a set of writes in an execution to be totally-ordered. We then show that a data-race-free program on BDSM has totally ordered writes.

Lemma 2.1 *If all writes in a program are seen by all processes in total order. and that total order obeys the program order of each process, then the program appears to be executing on a memory that is at least SC.*

This follows from the definition of sequential consistency. A total order of all writes seen by all processes is an execution that is a legal interleaving of the program-ordered writes of all processes. Since it is seen by all processes. there is one view of memory.

Lemma 2.2 *If an order can be established between any two (different) events in a system then the events are totally ordered.*

By the definition of total order. if $\forall x$ and $\forall y \neq x. \in \mathcal{H}$. x is before y or y before x then the elements of $\mathcal{H}$ are totally ordered.

Theorem 2.1 *A data-race-free program running on BDSM has totally ordered writes.*

Proof: Take *any* two distinct write events $a = w^x_{i,n}. b = w^y_{j,m} \in \mathcal{H}$. Either $a \ll b$ or $b \ll a$ in $\mathcal{H}$.

1. Consider $i = j$:

   - If $x = y$ or $s(x) = s(y)$, then a and b are ordered by the program order of p_i. Since $a \neq b$, $n \neq m$. Either $n > m$ or $m > n$. From axioms 2.1 and 2.3 and the definition of globally precedes, $a \ll b$ or $b \ll a$. The writes are ordered the same at all processes.
   - If $x \neq y$ and $s(x) \neq s(y)$ then a and b must have a synchronization operation s between them (definition 2.2).

     Suppose $a \prec b$ in h_i.

     - We have $a \prec s \prec b$.
     - From axioms 2.6 and 2.7, $u^{x,i,n}_{k,p} \prec s_{k,l} \prec u^{y,j,m}_{k,o}. \forall k \neq i$

- Since $\prec$ is transitive locally. $u_{k,p}^{x,i,n} \prec u_{k,o}^{y,j,m}. \forall k \neq i$
- From the definition of $\ll$. $a \ll b$.

Suppose $b \prec a$ in h_i.

- We have $b \prec s \prec a$.
- From axioms 2.6 and 2.7. $u_{k,p}^{y,j,m} \prec s_{k,l} \prec u_{k,o}^{x,i,n}. \forall k \neq i$
- Since $\prec$ is transitive locally. $u_{k,o}^{y,j,m} \prec u_{k,p}^{x,i,n}. \forall k \neq i$
- From definition 2.4. $b \ll a$.

2. Consider $i \neq j$:

- Then a and b must have a global synchronization operation. a barrier. s between them (definition 2.2). From definition 2.3. either
  - h_i contains $a \prec s \prec u_{i,l}^{y,j,m}$ and h_j contains $u_{j,k}^{x,i,n} \prec s \prec b$.

    From axioms 2.6 and 2.7 and the definition of $\prec$. $u_{p,k}^{x,i,n} \prec u_{p,r}^{y,j,m}. \forall p$.

    Therefore $a \ll b$.
  - or h_i contains $u_{i,l}^{y,j,m} \prec s \prec a$ and h_j contains $b \prec s \prec u_{j,k}^{x,i,n}$.

    From axioms 2.6 and 2.7 and definition of $\prec$. $u_{p,r}^{y,j,m} \prec u_{p,k}^{x,i,n}. \forall p$.

    Therefore $b \ll a$.

Theorem 2.2 *BDSM can provide the appearance of sequentially consistent memory to data-race-free programs*

Proof: The proof of theorem 2.2 follows from theorem 2.1 and lemmas 2.1 and 2.2.

By showing that, in this extreme case, BDSM can appear sequentially consistent to programs we have shown that anything computable on a sequentially consistent memory is

computable using BDSM. This is a powerful statement of the utility of such a weak system. In chapter 5 we show that our implementation of BSDM is consistent with these axioms and therefore provides BDSM coherence as presented above.

2.3 Programming Interface

In this section we present the programming interface for our experimental system. An example of the usage of this interface can be seen in the BDSM version of the Jacobi program in appendix A. The actual implementation of the BDSM system is presented in chapter 5.

Starting a BDSM computation is a two step process. First, it is necessary to have processes running on separate machines. Second, they must each call `dsm_init`. The function `dsm_startup` can be used to perform the remote invocations of the program on different machines. This call is not required. The group members can be started individually. Using either invocation method, one (and only one) of them should have the `s_flag` parameter set to a non-zero value. This process will act as the server during the group registration protocol. This registration is done using the startup function of the PBP reliable broadcast protocol, presented in chapter 3. Once all of the processes have registered, execution can continue.

A single process uses `dsm_seg_at` to create a shared memory segment with a unique given key. The flag parameter should be set to `DSM_CREATE`. Other processes can then use `dsm_seg_at` with the flag set to `DSM_JOIN` and the same key to attach to this segment once it has been created. The number of locations and the size should be specified by all processes.

The function `dsm_remove` deletes the given segment from the local DSM system. It only effects the single calling process. Others processes can continue to access that segment. When a process is finished using BDSM completely it calls `dsm_exit`. This call forces all writes to be disseminated and removes the calling process from both the BDSM system and the PBP communication system. If the caller has no active segments, `dsm_exit` is non-blocking and does not effect other processes except to remove references to this process.

Since BDSM uses a granularity smaller than a page, it cannot use the memory management to make accesses to shared memory transparent. Access to a BDSM memory location is made through the read and write functions. A process writing to a single location provides a segment id, a location number within that segment and a pointer to the value to be written. Making this a function call rather than an assignment is necessary to allow the BDSM system to see the update and propagate it to other processes. The function `dsm_bulk_write` can be used for efficiency when a number of contiguous writes are made. This causes all buffered writes in the segment to be sent and then sends a single update of all of the data in the range specified. Using this function is ideal for data initialization and for a number of programs that write data in blocks.

Reading locations in BDSM is done with the `dsm_read` function. If a program has enough synchronization then it can make reads transparent by requesting a pointer to a location in a segment. This pointer can then be used with the subscription operation to access individual locations directly. However, since there is then no control over when a user process accesses a location it is important that there be no write-write or read-write data races for that location. We have found that this is common to many parallel numerical applications. This is not a limitation imposed by BDSM as it is necessary in any shared

memory environment. Barriers and locks are used between writes accesses and reads to prevent these data races.

2.3.1 Initialization and functions

```
int dsm_startup(int *argc, char **argv);
```

The `dsm_startup` function can be used to start programs on different machines. The command line arguments following a "`--`" delimiter are read by the BDSM system. The actual parameters include the number of processes to start, whether or not to create remote `xterm` windows on the local display, and whether to use `ssh` or `rsh` to make the remote connection. Most of these arguments can be specified in a configuration file. This call is designed to make the BDSM system easier to use. However, it does not have to be used to start the system. Each process may be started by hand. On success `dsm_startup` returns zero. On any error it returns a negative value.

```
int dsm_init(int *numprocs, int s_flag);
```

This is the primary initialization routine for the BDSM system. Each process in a computation must call this function. One and only one of these calls should have the `s_flag` value non-zero. This one process will be the server for the group registration routine. The `numprocs` parameter is set by BDSM to the number of processes in the group after registration is complete. On success the calling process' id within the group is returned, a value between 0 and `numprocs` - 1. A negative value is returned on an error.

```
int dsm_seg_at (int numlocs, int locsize, int dsm_key, int dsm_flag);
```

Once BDSM had been initialized using `dsm_init` separate segments of shared memory are created using `dsm_seg_at`. The first two parameters `numlocs` and `locsize` specify the number of locations and the size of each location, respectively, that this segment will have. The `dsm_key` is used to uniquely identify different segments. Since only one process creates a segment, other processes need to use the same key as the creator to join that segment. The final parameter, `dsm_flag` is used to either create (`DSM_CREATE`) a segment using the given geometry values and key or join (`DSM_JOIN`) an already created segment with the same key as the one supplied in the call. When joining a segment the `numlocs` and `locsize` should match those given by the segment's creator. On success a valid `dsm_id` descriptor is returned. On error a negative value is returned.

2.3.2 Memory Access Functions

```
int dsm_write(int dsm_id, int location, void *value, int rel_flag);
```

BDSM processes interact with the shared memory through several routines. The first is the basic write function `dsm_write`. This function takes a valid `dsm_id` descriptor (as returned by `dsm_seg_at`). It then writes the value pointed to by `value` to the given `location` in the specified segment. The final argument allows a process to issue a write that bypasses the reliability protocol for message delivery. If `rel_flag` is set to `DSM_WRT_REL` the normal reliable broadcast will be used for the update associated with this write. If `rel_flag` is set to `DSM_WRT_UNREL` then the associated update will be sent as a standard UDP datagram. This write will take effect locally, but may or may not be seen by all of the other processes. The return value is zero on success, negative on error.

```
int dsm_bulk_write(int dsm_id, int location, void *value, int num_writes,
    int rel_flag);
```

Since it is often the case that a process, especially when initializing shared data, writes to a number of contiguous locations, BDSM provides a bulk write function. This function is similar to `dsm_write`. However, it writes `num_writes` locations from `value` into the given segment starting at `location`. These writes will be grouped and sent as an update that takes advantage of this contiguity. Because less bookkeeping information is required, more data can be sent in fewer messages. Additionally, the overhead of multiple function calls is avoided. The return value is the number of locations written on success, a negative value on error.

```
void* dsm_read(int dsm_id,void* value, int location);
```

Memory locations are accessed for reading using the `dsm_read` function. This routine reads the value in `location` of the given segment into the memory space pointed to by `value`. It returns `value` on success, NULL on error.

```
void *dsm_ptr_read(int dsm_id, int location);
```

Since, in BDSM, reads are local operations, memory locations can be accessed for reading directly. This routine returns a pointer to the internal BDSM data space were `location` is stored in the given segment. This pointer can then be used as an array of the appropriate data type. However, it should not appear on the left of an assignment statement. On error, NULL is returned.

2.3.3 Synchronization Functions

```
int dsm_barrier(int b_id, int *num_procs);
```

This function performs a multi-process barrier. Each process must make a call to this function with the same identifier value, as given by `b_id`. The parameter `*num_procs` specifies the number of other processes required to cross the barrier. If this value is NULL then the system performs a total barrier. Before blocking for the barrier the BDSM system will flush all of the caller's buffered updates to ensure they are seen before the barrier. It returns zero on success and a negative value on error.

```
int dsm_lock_acquire (int locknum);
```

There are two functions that deal with locks. The first `dsm_lock_acquire` is used to acquire the given lock, `locknum`. The function returns zero on success and a negative value on error. The return value should be checked since, on error, mutual exclusion is not assured. It is considered an error to acquire a second lock without releasing the first.

```
int dsm_lock_release(int locknum);
```

This is the complement to the previous function. It releases the previously acquired lock, `locknum`. It is considered an error for a process to release a lock that it is not currently holding. The release routine returns zero on success and a negative value on error.

2.3.4 Clean-up and Exit Functions

```
int dsm_remove(int dsm_id);
```

A process removes a locally attached segment with the function `dsm_remove`. The segment given by `dsm_id` is removed from the local memory and no more incoming updates for this segment will be handled. The descriptor is then invalid for further BDSM operations, unless returned by a subsequent `dsm_seg_at` call. Before returning, all of the calling processes updates for this segment must be delivered to other group members. It returns zero on success and a negative value on error .

```
void dsm_exit();
```

The function `dsm_exit` removes the calling process from the BDSM system. It will first remove any attached segments, and then close the communication channels to the BDSM group.

2.3.5 Configuration File

Some of the functionality of the BDSM system can be controlled by a configuration file that is read when the system starts. Each process should have access to identical copies. This text file consists of a number of flags that determine the behavior of the BDSM system. Most are used by the `dsm_startup` function. The location of the file is either the current directory or the directory defined in the environment variable `DSM_WORKING_DIR`.

`DSMEXECPATH` This should be set to the full directory path to where the binary lives so that it can be executed remotely.

`XTERMCOMMAND` This line should be set to the command to run to generate a terminal.

`MACHLISTFILE` This should be the full name of a file listing machines to start remote jobs on.

`USEXTERMS` This flag tells the system to start remote terminals on the root machine.

`USESSH` This flag specifies that ssh should be used instead of rsh to make remote connections.

`BUFFERWRITES` This flag allows the user to control whether or not the system buffers writes. If it's set to zero, the system will send each update as the write call is made.

2.4 Conclusions

We have presented a new weak DSM model. This model is based on using broadcast to supply updates to replicated copies of the shared space. Our model overcomes the problems of similar, non-causal memories by using message-passing for synchronization operations. These synchronization operations provide enough order that a program that is correctly written can see a sequentially-consistent memory model rather than the weaker BDSM model. We also presented the basic programming interface for our system. In the next chapter, we look at the PBP protocol that supplies the program-ordered, reliable broadcast which forms the basis of our implementation of BDSM.

Chapter 3

The Pipelined Broadcast Protocol

The previous chapter presented a new model for DSM, called BDSM. BDSM is inherently designed to use broadcast to disseminate updates. Due to the potential for message loss using UDP on an Ethernet segment, an implementation of BDSM requires some form of reliable broadcast protocol. Therefore we have developed a reliable broadcast protocol called Pipelined Broadcast Protocol or PBP. It guarantees that all messages sent are delivered and that they are delivered in the order sent. While there have been other reliable broadcast protocols developed, our system is different from the previous examples in several ways. First, PBP is designed exclusively to use hardware broadcast. We have designed it specifically for a common networked environment. Second, we provide only source order. Messages from any process are delivered in the order sent. There is no global ordering. Causal or atomic ordering could be implemented on top of PBP, if required. Third, PBP is a low-level protocol, not a general collection of services. We provide a minimal interface consisting of send and receive. Fourth, our primary goal is to make the common all-to-all communication patterns used in many parallel programs as efficient as possible on a net-

work cluster computational platform. We use our system in place of other network-based message passing systems such as the MPICH[45] implementation of MPI. or a collection of TCP connections.

The primary function of the PBP system is to ensure the delivery of every message sent. Since it uses a modified windowing protocol. and thus keeps track of message sequence numbers. ensuring process order requires little extra work. Without the possibility of retransmission. messages cannot be received out of order: they can only be omitted. The Ethernet acts as a serializing influence. Only one message can be on the wire at a time. However, messages that are lost will leave gaps in the order. If they are subsequently retransmitted. due to a timeout. they will then arrive after messages with higher sequence numbers. This would be a violation of FIFO delivery order. So each process maintains a buffer for each other process. Messages received are placed in this buffer and only delivered to the application when all preceding messages. from the same sender. have been received and delivered. This allows the system to be seen as a collection of FIFO queues. or pipelines. at each process. In an n process system there is one outgoing queue and there are $n-1$ incoming queues at each process. When the message at the head of any incoming queue has a sequence number equal to the expected sequence number from the corresponding process. it is eligible to be removed and placed on a general delivered queue. An application process can then consume items from this queue as they become available.

For the PBP system we assume all of the processes are known to each other at startup. This means the processes must know the total number of processes in the group and a common port number. This is done through a group registration phase as part of initialization. One process, called the server, which will have process id zero, receives messages on the

common port. All other processes send a message to the server and await a confirmation in return. Once the confirmation is received each process broadcasts a message with sequence number zero. When this message has been acknowledged by all other processes the protocol has started. Control is returned to the application process. and it may then begin sending and receiving messages. Since we are not dynamically making connections to long running systems we do not need a true three-way handshake to initiate the protocol.

Processes running on networked workstations can fail. The host workstation may go down. be rebooted or corrupted is some way that destroys the process that is a member of a given computation. When this occurs. PBP will timeout and take steps to determine if the process has indeed failed. In order to ensure that the remaining processes can continue to send messages we need to remove dead process from the acknowledgment protocol. Since we are using a simple. flat network topology. failure detection is not as complex as it might be in other domains. There is no way for more than one process to become partitioned from the rest. A single workstation may become disconnected but that is. in effect. a crash failure. Our system assumes crashes will be rare. but also behaves in a pessimistic manner regarding declaring a process dead. Once a process is declared dead it is assumed to always acknowledge every messages as soon as it is sent. This way the remaining processes will continue to make progress. We do not handle potential recovery or returning a process to the group once it has been removed.

We have developed and implemented two versions of PBP. The first uses a positive acknowledgment protocol where the sender retransmits messages if it has not received acknowledgments in a certain amount of time. The second uses a negative acknowledgment protocol where receivers request resends of missing messages. In the following sections. we

present a detailed look at each version of PBP.

We chose to use Ethernet broadcast addresses for this implementation. The other option would be to use IP Multicast[34, 86]. IP Multicast would allow multiple groups within the PBP system. Each process could join multicast groups it is interested in and in theory only be interrupted by network packets sent to those groups. It would also allow for systems to be on different segments if connected by multicast aware routers. Since our goal was specifically the hardware broadcast we didn't need this latter benefit. As currently implemented the system is designed to have a single communication group. In chapter 7, we discuss allowing smaller divisions of the computation to improve scalability. It is there that IP Multicast would be most useful. However, the network interfaces we are using, 3Com 3c509 cards, only have binary filtering[44]. Therefore, all processes would still need to have a software interrupt to handle all packets to determine if they are for multicast groups the local machine is a member of. The benefits of selectively interrupting only those machines that have joined a group is lost with this particular hardware.

3.1 Positive Acknowledgment Protocol

The first version of PBP (PBP1), is a positive acknowledgment protocol. Such a protocol requires some form of response from receiver to sender acknowledging receipt of each message. It bases the retransmission of potentially lost messages on a timer expiring at the sender before this response has been received. PBP1 is an extension of a standard window protocol with delayed acknowledgments. Rather than a single expected sequence number from a single connected sender, each process maintains a vector of expected sequence num-

bers. At process i the jth element of this vector corresponds to the expected sequence number from process j. Processes maintain a vector of incoming acknowledgments as well. The minimum value across each process' vector is the current base of the window at that process. All messages with sequence numbers less than this minimum have been received and acknowledged by all processes and thus no longer need to be buffered at the sender. Each message sent by process i will have a single sequence number. It will also have a vector of acknowledgments. When process j receives a message from process i, it uses the value of the ith entry of this vector as an acknowledgment for its messages that have sequence numbers less than or equal to that value. This potentially increases the minimum value of j's acknowledgment vector and allows the window at process j to slide upwards.

3.1.1 Protocol Presentation

Assume a set of n nodes, numbered $0..n-1$ on a broadcast medium network. Each node runs a user process that requires reliable FIFO broadcast service and a PBP layer that provides it. The PBP layer is designed to operate as a middle layer between a user process and the broadcast functions of a network. PBP communicates with a user process by way of two queues of user message data. The `send_q` is used when the user calls the send function to ensure sending a message is non-blocking at the user level. This queue also ensures that message sending events by the user are handled in FIFO order by the PBP system. Messages to be consumed by the user process are put in the `recv_q` by the PBP layer. The user process can then dequeue this data as it needs to. These two queues obey the usual semantics. For the discussion of the protocol we are not interested in the specifics of the user process. We are concerned with getting messages in order and placing them on the

recv_q. Once this is done for a message it is beyond the concern of PBP. Therefore, when we talk about process p_i we mean the PBP layer at node i.

Local data:

int *ws*: *// window size*
msg $in\_buffers[0..n-1][ws]$ of messages: *// to reorder messages*
msg $out\_buffer[ws]$: *// Outgoing buffer, sent but not acked*
queue *send_q*, *recv_q*: *// hold user messages*
int $exp\_seqno[0..n-1]$: *// exp_seqno[i] is next seq number to send at i*
int $window\_base[0..n-1]$: *// bases of other's windows*
int $wb = \min_j(window\_base[j] \forall j \neq i)$: *// acks, wb is what can be base of window.*
int *curr_base*: *// exp_seqno[i] - curr_base == number of outstanding messages*
bool *ack_flag*: *// initially false*

Figure 3.1: Local Data for PBP1

Figure 3.1 lists the state that is used at each process i during normal operation. The window size is determined at runtime and is stored in *ws*. For PBP1, this is usually set to 16 messages. Once the protocol starts up, each process will have sent, and acknowledged, a message with sequence number zero. User messages begin at one. The highest sequence number for which all acknowledgments have been received is *wb* which is, at all times, the minimum value in the vector window_base, excluding the ith element. The base of the local window is defined by *curr_base*. Since *wb* can change based on messages received these two variables are separate. However, each time the window is adjusted *curr_base* will be set to equal the value of *wb* at that time. The ith element of the vector *exp_seqno* holds the next sequence number to send. The last message sent is *exp_seqno*[i] - 1. The number of pending messages is, therefore, *exp_seqno*[i] - *curr_base*. These pending messages are held in a message buffer, *out_buffer*[*ws*], which is indexed circularly modulo *ws*. The next

buffer location to use is $exp\_seqno[i]$ mod ws. To reorder incoming messages, each process maintains an array of message buffers, $in\_buffers[0..n-1][ws]$. Message from process j are placed in the appropriate buffer location of $in\_buffers[j]$. These buffers are indexed in the same manner as the *out_buffer*, except they use the sequence number of the arriving message modulo ws as the determinant. The expected sequence numbers from other processes are stored in $exp\_seqno[0..n-1]$. Each of these numbers serves as both the next sequence number expected and an acknowledgment for all earlier messages. For example, at process p_i, $exp\_seqno[j]$ is the next message p_i should receive from p_j and all messages with sequence number $s < exp\_seqno[j]$ have been delivered at p_i.

Definition 3.1 *Delivered: We say a message from some p_j that has been received by p_i from the network and enqueued on the local recv_q has been* delivered *at p_j.*

```
User Calls:
    send_msg(data) {
        enqueue(data,send_q);
    }
    data recv_msg(){
        While (recv_q is empty ) nop;
        return (dequeue(recv_q));
    }
```

Figure 3.2: User calls to PBP1

An application program communicates with PBP, in essence, through two functions, shown in figure 3.2. To send a PBP message a call is made to the send_msg function. The application data to be sent is passed as a parameter. The message is simply appended to

the send_q. To receive a message a blocking call is made to recv_msg which returns a data message when one is available on the recv_q. The use of these queues ensures messages to and from the PBP layer are in FIFO order. All messages sent by the application are handled by the local PBP process in the order they are enqueued. Similarly, all messages delivered by the PBP layer are handed to the application in FIFO order.

Message $m = (t, i, n, e, data)$ where

t = message type: PLAIN_ACK or ORDINARY
i = sending process number
n = sequence number of this message
e = vector of acks, highest sequence number delivered for each process at i
$data$ = user level message.

Figure 3.3: Message format for PBP1

Each PBP1 message consists five components, as shown in figure 3.3. The message type, either PLAIN_ACK or ORDINARY, defines how the message will be handled. User messages are type ORDINARY and will have non-null *data*. The other type, PLAIN_ACK is used when the protocol needs to explicitly send an acknowledgment. This occurs when there are no outgoing messages on which to piggy-back the acknowledgments. In this case, *data* will be NULL. The sending process's process number, i, is included in each message. Combined with the sequence number, n, this uniquely defines each message. The vector e is a copy of p_i's local exp_seqno vector. These are the piggy-backed acknowledgments.

To define the protocol we will use a form of guarded command notation where each guard that is enabled may be executed at any point. The protocol is essentially a forever loop, performing whichever actions are enabled as possible. All of the instructions in each guard are executed atomically. In general, they may not be interleaved. This is a little

more strict than is actually necessary in practice, where access to critical sections can be synchronized when using multiple threads of execution. Figures 3.4 and 3.5 show the actions for the receiving and sending sides of the protocol, respectively, for process i.

```
Receiving Actions:
    1 receive message m = (t, j, n, e, data) from network do
        if (i == j ) continue;
        window_base[j] = max(window_base[j], e[i] - 1);
        if (t == PLAIN_ACK) continue; // We already got ack info
        if (n < exp_seqno[j])
            set ack timer;
            continue;
        if (in_buffers[j][n mod ws] == NULL)
            in_buffers[j][n mod ws] = m;
    od;
    // Pass in order messages to application through recv_q
    2 while (∃j ≠ i : in_buffers[j][exp_seqno[j] mod ws ] != NULL) do
        enqueue(in_buffers[j][exp_seqno[j] mod ws ]-> data, recv_q);
        in_buffers[j][exp_seqno[j] mod ws]= NULL;
        exp_seqno[j] ++;
        set ack timer
    od;
```

Figure 3.4: PBP1 Receive Actions

The receipt of a message happens with action 1. When a message is available from the network at p_i the message can be handled. The first step is to ignore self-sourced messages. Since this is true broadcast, each process usually receives each message sent, including those sent by itself. The next statement applies the acknowledgment from p_j. The value of $e[i]$ is the next sequence number p_j is expecting from p_i, so $e[i] - 1$ is the last message delivered

at p_j from p_i. All messages with sequence numbers less than or equal to $e[i] - 1$ have been successfully delivered to p_j. If this is a PLAIN_ACK message, p_i is done with it once the acknowledgment has been applied. If n is less than the expected sequence number from p_j, this is a retransmission p_i does not need. In this case, p_i sets the ack_flag variable so it will send an acknowledgment in case p_j is retransmitting messages due to a lost acknowledgment. Then, p_i is done with the old message. Finally, if this is a new message we place it in the in_buffers location for process j, based on n modulo ws.

Action 2 is responsible for passing messages to the user level. It is enabled when the in_buffers location for the base of any other p_j's window has a valid message in it. In this case, the *data* from that message m is appended to the recv_q. The buffer location is then cleared by setting it to NULL. The base of p_j's window, exp_seqno[j], is incremented to reflect the delivery of message n from p_j. Additionally, p_i sets the ack_flag so it will send a PLAIN_ACK message stating this fact if needed. If action 2 is performed each time action 1 is, then it can be simplified to only consider the sender, p_j, of the message that triggered action 1.

In order to reclaim out_buffer spaces and slide the window upwards, p_i executes action 3. If the current base of the window is less than or equal to the minimum value that has been acknowledged, represented by wb, p_i clears the corresponding out_buffer location. Then, it increments *curr_base*. This has the effect of lowering the number of outstanding messages and could therefore enable more messages to be sent. Finally, the resend timer is reset to prevent it from attempting to retransmit messages if there aren't any outstanding.

A timeout occurs when one of the timers, either the ack timer or the resend timer, expires. When this happens action 5 is enabled. The resend timer, in PBP1, triggers

Sending Actions:

//Move window base upwards and clear buffer locations if possible

3 while ($curr\_base <= wb$) do

clear $out\_buffer[curr\_base \bmod ws]$:
$curr\_base{+}{+}$:
clear resend timer:

od:

//Send messages if there are some and there is buffer space

4 while (!empty($send\_q$) $\wedge$ $exp\_seqno[i]$ - $curr\_base < ws$) do

m = (ORDINARY, i, $exp\_seqno[i]$, $exp\_seqno[0..n-1]$, dequeue($send\_q$)):
$out\_buffer[exp\_seqno[i] \bmod ws] = m$:
send m to network:
$exp\_seqno[i]{+}{+}$:
set resend timer:
clear ack timer :

od:

Figure 3.5: PBP1 Send Actions

the retransmission of all of outstanding messages. The timer gets set whenever there are any messages outstanding. This is the method by which message loss is overcome. If a process does not receive the appropriate acknowledgments then the resend timer will expire and messages will be retransmitted. The second timer is the ack timer. It is set when new messages are received. Since acknowledgments are delayed, a process must send a PLAIN_ACK at times. This timer specifies when this happens. It needs to be set less than the resend timeout so that PLAIN_ACKS will be sent before messages are retransmitted to reduce extra messages.

```
Timer Event:
    5 timeout do
        if (resend timer expired) {
            for each m = (t.i.n.e.data) in out_buffer
                send m = (t.i.n.exp_seqno[0..n − 1].data):
            clear ack timer:
            reset timer:
            }
        if (ack timer expired){
            send m = (PLAIN_ACK. i.exp_seqno[i].exp_seqno[1..n − 1].NULL):
            clear ack timer:
            }
    od:
```

Figure 3.6: PBP1 Timer Event

3.1.2 Formalism and Proofs

In order to use PBP for higher level applications. it is important to show that the PBP protocol provides two important properties. These are that all messages sent are delivered at all other processes once. and only once. and that these messages are delivered in the order of the send events in the sending process. We call these *PBP Property 1*(definition 3.2) and *PBP Property 2*(definition 3.3) respectively.

Definition 3.2 *PBP Property 1:* $S_i^k \implies D_{i,j}^k . \forall k, j$. *Each message sent by any process* i *is delivered once and only once at all process* $j \neq i$.

Definition 3.3 *PBP Property 2: Messages are delivered in the order sent. For any two distinct messages* m *and* n. *sent by* p_i *with sequence numbers* h *and* k. *if* m *is sent before* n *then* m *is delivered before* n *at all* j.

In this section we prove that the protocol provides these two properties. We start by defining a set of axioms drawn from the protocol.

Throughout, we use the following events:

- S_i^k = send of message with sequence number k by p_i.
- $R_{i,j}^k$ = receipt of message with sequence number k sent by p_i at p_j.
- $D_{i,j}^k$ = delivery (see definition 3.1) of message k sent by p_i at p_j.
- $A_{i,j}^k$ = acknowledgment, at p_i, from p_j for message k. If $A_{i,j}^k . \forall j \neq i$ then message k has been delivered at all processes.

Additionally, we use P to represent the set of all processes, $(0..n-1)$, when needed. It is assumed in most cases.

Axiom 3.1 $S_i^k \implies S_i^h . \forall h < k$.

Messages are sent with sequence numbers in ascending order.

Axiom 3.2 $R_{i,j}^k \implies S_i^k . \forall i, j \neq i$.

Messages cannot be received before being sent.

Axiom 3.3 $R_{i,j}^k \wedge D_{i,j}^h . \forall h < k \implies D_{i,j}^k$.

Messages received are delivered if all previous messages from the same sender have been delivered.

Axiom 3.4 $S_i^0 \wedge R_{i,j}^0 \wedge D_{i,j}^0 . \forall i, j \neq i$.

Initially, due to the nature of the start up sequence, message number 0 is sent, received and acknowledged by each process.

Axiom 3.5 $D^k_{i,j} \wedge R^h_{j,i} \wedge (S^h_j$ *happened after* $D^k_{i,j}$ *became true* $) \implies A^k_{i,j}$.

A delivered message is acknowledged by later messages sent by the process at which the message was delivered.

Axiom 3.6 $S^k_i \implies R^k_{i,j}. \forall j \in P' \subseteq (P - i)$. *Note:* $P' = \emptyset$ *is possible.*

Messages sent may be lost.

Definition 3.4 *Network Liveness Axiom: We make the assumption that the network has not completely failed. If* p_i *sends the same message* k *some finite number times each other process will receive message* k. *That is, the probability of a message being lost is low enough that the probability of not getting a message to each process, given a finite number of retransmissions, approaches zero.*

Axiom 3.7 $S^k_i \wedge \exists j. \neg A^k_{i,j} \implies R^k_{i,j}$.

Message that are sent, but not acknowledged will be retransmitted.

Theorem 3.1 *PBP is consistent with axioms 3.1-3.7.*

Proof: We show this by examining each axiom in turn.

- Axiom 3.1 follows from "m = (ORDINARY, i, $exp\_seqno[0..n-1]$, $exp\_seqno[i]$, dequeue($send\_q$))" and "$exp\_seqno[i]$++" in action 4.
- Axiom 3.2 is self-evident.

- Axiom 3.3 follows from the condition for action 2 and the increment of $exp\_seqno[j]$. The reverse holds as well: $D^k_{i,j} \implies D^h_{i,j}.\forall h < k$. And, similarly, $R^k_{i,j} \wedge \neg(D^h_{i,j}.\forall h < k) \implies \neg D^k_{i,j}$

- Axiom 3.4 holds or the protocol has failed to start.

- Axiom 3.5 If a message is delivered at p_j and p_j subsequently sends a message, which is received by p_i, then p_j has acknowledged message k from i. The "happened after" relation here is well defined because it is local to p_j and is based on the program order of p_j.

- Axiom 3.6 Due to the possibility of messages being lost on the network, when p_i sends a message it will be received by some subset of the other processes.

- Axiom 3.7 From the Network Liveness Axiom, definition 3.4, message k will be received after some finite number of resends. From action 5, messages not acknowledged will be retransmitted.

Therefore the theorem holds.

We would like to show that PBP1 provides the guarantee that all messages sent by some process p_i are delivered at all other processes. This property is called PBP property 1 (P1), see definition 3.2. The basis of the proof is lemma 3.2, which shows that messages that are not acknowledged will be retransmitted, and, because the probability for loss is low enough, given enough resends each message will be received.

Lemma 3.1 $D^k_{i,j} \rightsquigarrow {}^1 A^k_{i,j}$

[1] *This is the temporal leads to. Informally, $A \rightsquigarrow B$ means if A then at some finite time later B must also be true.*

All messages delivered get acknowledged. eventually. The proof is based on two cases. If p_j sends a message subsequent to $D^k_{i,j}$ axiom 3.5 applies and $A^k_{i,j}$ follows. Otherwise. a timeout occurs at p_j in action 5 and a PLAIN_ACK is sent. This also acknowledges k. If either message is lost it will get retransmitted. This happens either due to j not receiving an acknowledgement to its subsequent message or to j receiving another copy of k from i.

Lemma 3.2 $S^k_i \implies R^k_{i,j}. \forall j \in P - i.$

The proof of lemma 3.2 is based on axiom 3.6. The axiom can be divided into two cases. These are $P' = (P - i)$ and $P' \subset (P - i)$. That is. a given message is sent and it reaches all other processes. or it fails to reach at least one of the other $n - 1$ processes.

$S^k_i \implies R^k_{i,j}. \forall j \in P' \subseteq (P - i)$ from axiom 3.6.

- $P' = P - i$: $S^k_i \implies R^k_{i,j}. \forall j \in P - i$ follows directly from axiom 3.6.
- $P' \subset P - i \implies \exists j. \neg R^k_{i,j}$. For each such j:

  1. $\neg R^k_{i,j} \implies \neg D^k_{i,j}$ from axiom 3.3
  2. $\neg D^k_{i,j} \implies \neg A^k_{i,j}$ from axiom 3.5
  3. $S^k_i \wedge \exists j. \neg A^k_{i,j} \implies R^k_{i,j}$ from axiom 3.7 and the Network Liveness Axiom.

Therefore $S^k_i \implies R^k_{i,j}. \forall j \in P - i.$

Theorem 3.2 *PBP Property 1 holds for PBP1*

For the "delivered once" case we prove by induction on k.

- Base case $k = 1$:

1. $S_i^1 \implies R_{i,j}^1 \forall j \in P' \subseteq (P - i)$ from axiom 3.7.

2. $S_i^1 \implies R_{i,j}^1 \forall j$ since $P' = P - i$ from lemma 3.2.

3. $R_{i,j}^1 \wedge D_{i,j}^0 \implies D_{i,j}^1$, from axioms 3.4 and 3.3.

4. $S_i^1 \implies D_{i,j}^1$, by substitution.

- Now assume $S_i^k \implies D_{i,j}^k$ for all k. show $S_i^{k+1} \implies D_{i,j}^{k+1}$

1. $S_i^{k+1} \implies R_{i,j}^{k+1} \forall j \in P' \subseteq (P - i)$ from axiom 3.7.

2. $S_i^{k+1} \implies R_{i,j}^{k+1} \forall j$ since $P' = P - i$ from lemma 3.2.

3. $R_{i,j}^{k+1} \wedge D_{i,j}^k \implies D_{i,j}^{k+1}$, from assumption and axiom 3.3.

4. $S_i^{k+1} \implies D_{i,j}^{k+1}$, by substitution.

To prove "only once" we rely on the uniqueness of messages. Each message has a unique identifier, its sequence number and sender's id. Once a message is received at p_j it will not be handled again. Consider message $m = (t, j, n, e.data)$. There is only one buffer location for message m at p_i. Once it is filled and all previous messages have been delivered, m is delivered. The expected sequence number for process p_j at p_i is set equal to $n + 1$. At this point no message numbered $\leq n$ will be handled, from action 1. If m is received again it will be ignored. If m is received a second time before it is delivered it will also be discarded because the buffer space it needs to go is occupied. So it will only be delivered once. In a practical windowing protocol with finite (and hence reused) sequence numbers there must be at least $2w + 1$ different sequence number, where w is the window size[90]. In PBP, we use a 16 bit sequence number, giving $2^{16} - 1$ unique sequence numbers. Window sizes used

are in the range of 2^4 to 2^7 messages, with 2^4 being the usual value for PBP1. Recycling sequence numbers is not a problem.

The second property we would like to establish is that PBP1 provides message delivery in FIFO order. This means messages sent by some process i are delivered at all j in the same order they were sent. This is essentially encapsulated in the use of integer sequence numbers. We show that, based on the axioms derived from the protocol, if messages are not delivered in the order sent there is a fundamental contradiction.

Theorem 3.3 *PBP Property 2 holds for PBP1.*

Proof by contradiction:

1. Assume m is sent before n and n is delivered before m at some j.
2. m is sent before $n \implies h < k$. From axiom 3.1.
3. n is delivered before $m \implies$ that at some point n is delivered and m is not. Therefore $D^k_{i,j} \wedge \neg D^h_{i,j}$.
4. $D^k_{i,j} \implies D^h_{i,j}, \forall h < k$, from axiom 3.3.
5. $m \neq n \implies h \neq k$ and $h < k$ we have a contradiction: $D^h_{i,j} \wedge \neg D^h_{i,j}$.

Therefore, if m is sent before n, m is delivered before n at all j.

We have shown that the protocol for PBP1 provides the service it claims to. Programs that use PBP1 can rely on it to deliver all messages, in the order sent, to each process in the group.

3.1.3 Implementation of PBP1

We have implemented the PBP1 system on the GNU/Linux operating system on Ix86 processors. Our primary lab consists of a 10Mb/s Ethernet network of 120Mhz Pentium systems. The PBP1 system is a user-level C library which uses the LinuxThreads[61] implementation of the POSIX threads standard[12]. Figure 3.7 shows the inter-relations of the components of PBP1. All of the threads run in the same user address space. The PBP1 protocol consists of two executing threads and interface functions. One thread handles incoming messages. The other is used as a periodic timer to handle retransmissions and delayed acknowledgments. The interface consists primarily of send and receive routines. Messages are delivered to the user thread through a shared message queue. Dequeuing a message can be a blocking action or a simple poll as specified in the function call.

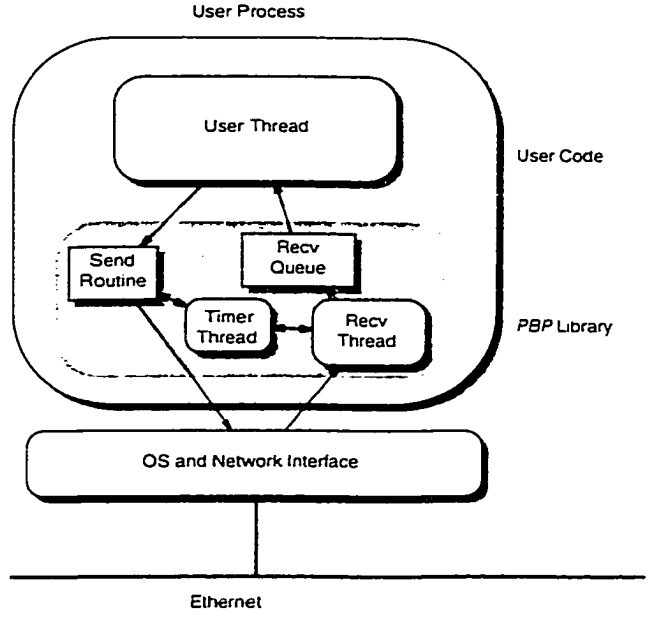


Figure 3.7: PBP1 Design Layers.

Our implementation of PBP is designed to provide discrete packets to the user. We do

not allow packets larger than the maximum transmission unit (MTU) of the Ethernet. 1500 bytes. Because PBP currently resides above the UDP layer this does not affect our protocol. Larger messages would be fragmented by the UDP layer and only delivered to PBP when reconstituted. Preventing fragmentation facilitates making a system that bypasses the UDP layer, which is a potential optimization for PBP. This restriction also allows us to know that each PBP message is exactly one Ethernet packet on the hardware.

The PBP system is designed to add little overhead to the UDP layer it is built on. To this end, we use a zero *extra*-copy technique. We take steps to ensure that data is copied no more than it would be using regular UDP communication. User processes allocate the space for each message to be sent and the system de-allocates this space when the message is successfully delivered. The user process can then build its messages in the same data space that will be used by PBP for the broadcast message. This whole packet is passed by reference to the send routine. Similarly, a received message is copied into a dynamically allocated memory region by the UDP `recvfrom` function within PBP. This message is then handled by reference, until it has been consumed and the user process de-allocates it. This helps to reduce the overhead of our system, which is important because it is an added layer in the protocol stack.

The timer thread handles signals from an interval timer which is set to go off periodically. When the timer expires, the thread checks the current state of messages that may have been sent but not acknowledged. If there are un-acknowledged messages that have expired, they are retransmitted, starting with the base of the window. It is possible that several may be lost in a row. Therefore, we currently retransmit the entire window of outstanding messages when a retransmission is required. The timer thread also determines if too much time has

passed since the local process sent a message, thus failing to acknowledge received messages. If this is the case, a plain acknowledgment message will be sent. As is usual with delayed acknowledgment protocols, a balance needs to be struck between acknowledgment timeouts and retransmission timeouts to try to minimize the number unnecessary retransmissions. We have not implemented a dynamic timeout system for two reasons. First, the notion of round-trip time, which is the basis of dynamic timeouts, is somewhat ill-defined in a broadcast paradigm. Also, the simple topology of the networks we use should not be subject to as wide a variation in latency as a general TCP connection.

PBP1 uses a form of delayed acknowledgment windowing protocol. The aim is to reduce the number of empty acknowledgment messages. Assume an n process system with window size w. In a worst case scenario, where there is one sending process and there is a significant (greater than 200 millisecond) pause between each message, our system requires $n-1$ separate acknowledgments, one from each receiver for each message. However, if there is such a small amount of communication these extra messages should not be a problem. If there is no pause between messages from the single sender then there are on the order of $n-1$ acknowledgments per w messages. An ideal situation consists of all n processes continually sending messages. In this case, due to piggy-backing, no extra acknowledgment is sent. A more normal situation is where all processes are periodically sending messages. In this case, there will be a few acknowledgments needed by any processes that complete their sends before others. At the least, there will be $n-1$ as the other processes acknowledge the last message in the batch, unless the next communication batch starts before the timer expires. This is the communication paradigm we are targeting. Many barrier-based, parallel computations exhibit this behavior. For these programs, PBP1 tends toward, but does not

reach. a minimal number of extra acknowledgments.

There are several performance variables that may be changed in the PBP1 implementation. First. the size of the window might have an impact on the performance of a given program. Some communication patterns might be handled more efficiently with different window sizes. Secondly. the timeout for retransmitting or acknowledging messages can be changed. Currently. we use a timeout of 150 milliseconds for unsent acknowledgments and twice that for retransmitting messages. This keeps the timer thread from executing too often. but makes handling lost messages somewhat expensive. With the extremely low loss rate on modern Ethernets we feel this is justified. Messages are lost on the order of one per several thousand messages when sent as fast as possible. For messages with some. even small. amount of time between sends this rate is even lower. Traditional windowing protocols use a dynamic time out that tracks round-trip latency. With broadcast and a collective acknowledgment protocol. round-trip latency is not as clearly defined. Additionally. since the topology we are using is flat. the variation in message delivery time should be very small.

3.2 Using Negative Acknowledgments

Positive acknowledgments require a delay before message loss is detected and messages can be retransmitted. This is true even though the receiving process likely detects the loss as soon as the next message from the same sender arrives. The lost message will leave a gap in the sequence. In this case. it is possible that. rather than waiting for a timeout. the receiver can explicitly request a resend of the missing message. This is done with a

negative acknowledgment. The second version of the PBP protocol is designed to increase throughput and make recovery from missing messages faster than in PBP1.

3.2.1 Protocol Presentation

The overall structure of PBP2 is similar to PBP1. The user interface and the data locations are the same as in figures 3.2 and 3.1. Messages have the same format as figure 3.3 with the addition of NACK as a message type. However, the actions performed are different and there are several new ones to deal with nacks. The biggest difference is that the receiving process now has two distinct running states: normal and need_resend. When a process detects a missing message it enters the need_resend state. In this state, it can only execute certain actions that cannot lead again to need_resend state. No messages except NACKs, PLAIN_ACKs and messages from the process whose message is missing are handled.

First we look at the normal state actions. The sending actions (3 and 4), are the same as in PBP1 (figure 3.5). There is still a window and acknowledgments need to be applied in order to clear buffer spaces. A process must keep all messages it has sent until they are acknowledged because there is, until that time, the possibility another process will request a resend. We use the send timer, which should have a longer timeout than for PBP1, to ensure processes can continue. It is possible for the last message in a batch to be lost. In this case, the receiving processes will not see a gap in the sequence numbers because a later message is not sent. When the resend timer expires, a plain ack is sent to other processes, with the sequence number set equal to the last message sent. This will allow receivers to see that a message was missed and send a NACK if needed.

Action 1, receiving and handling messages is necessarily more complicated than in PBP1.

Action 1. in figure 3.8. shows how PBP2 receives a message in the normal state. The data field is used as a list of requested resends. The first element is used for the target process id. The remaining elements list the sequence numbers needed. While in the need_resend state. the protocol cannot handle any messages that could potentially trigger a transition to need_resend. instead these messages are put on a message queue. If there are any messages in this queue then they are received instead of a message from the network in action 1. Once a message is received or dequeued. it is handled. If the type is NACK and if it is targeted to this process. the requested messages are retransmitted. Then the message sequence number is compared to the sender's window and expected sequence number. If the message is in the window and its not the expected message then a NACK message is sent and p_i enters the need_resend state. The message is stored in the reordering buffer for later delivery. Action 2 is the same as in PBP1 (figure 3.4). Messages are delivered from the reordering buffers in the same manner as PBP1.

The PBP2 system uses three timers: a resend timer. a nack timer and an ack timer. The resend timer is set whenever there are outstanding messages. Unlike PBP1. when this timer expires it does not signal a resend of all of these outstanding messages. rather it causes a PLAIN_ACK message to be sent. This will have the sequence number of the last regular message sent by this process. This will be seen by other processes and can trigger a NACK if the last message was lost. Otherwise. it would be possible to lose a message and have the receivers not see a gap because there was never a later message. The second timer is a nack timer. It is set when the process enters the need_resend state. It serves to ensure a process does not remain in this state too long by triggering a resend of the NACK in case it. or any of the retransmitted messages got lost. The final timer is an ack timer. It serves

to ensure a plain ack is sent if an ack cannot be piggybacked on an outgoing message in time. In this way, processes can perform garbage collection and clear buffers during a lull in message-passing. Figure 3.9 shows the pseudo-code for the timer action in PBP2.

The major difference between the operation of PBP2 and that of PBP1 is in the dual state mechanism. When a process detects a lost message, either by seeing a gap in the sequence of regular messages or by getting a PLAIN_ACK with a sequence number higher than expected, it enters the need_resend state. It does this by sending a NACK requesting a resend of the missing messages to the sending process. In this state, it handles only those messages it needs to fill in the gap and any NACK messages from other processes. Other messages are enqueued and handled after a transition back to *normal* state. Figure 3.10 shows the pseudo-code for message receipt in the need_resend state. Acknowledgments from the incoming messages are checked. Then, messages from *nack_target* are handled if they are in the range of missing messages. If not, they are enqueued for later inspection. NACK messages sent by other processes are also handled. Actions 2 and 3, sending messages and clearing buffer space, occur in both normal and need_resend states.

3.2.2 Formalism for PBP2

The axioms from section 3.1.2 apply to PBP2. The basic functionality of the two protocols is the same. The difference comes in how message loss is detected and how messages are retransmitted. Receipt of messages and delivery of messages to the user-level are the same.

PBP2 differs from PBP1 primarily in the triggering message for re-sending messages. To show that P1 and P2 hold for PBP2, it is necessary to show that messages that are lost are retransmitted. That is, that PBP2 is consistent with axiom 3.7. The other axioms hold

because the related parts of the protocol did not change. Axiom 3.7 states that messages that are sent but not acknowledged are resent. Due to axiom 3.5 this means a message was not received. PBP2 will request a re-send until the message is received and delivered. Due to the Network Liveness Axiom this request must eventually be received by the original sender and the message will be resent. Therefore axiom 3.7 holds for PBP2.

Theorem 3.4 *PBP Property 1 holds for PBP2*

This follows directly from theorem 3.2. since the axioms are valid for PBP2.

Theorem 3.5 *PBP Property 2 holds for PBP2*

This follows from theorem 3.3. The axioms for ordering messages. those involving sequence numbers and delivery. are unchanged for PBP2.

3.2.3 Implementation

The implementation of PBP2 is extended from that of PBP1. We use the same systems for both libraries. However. there are two major differences. First. since the timeouts in a negative acknowledgment protocol are based on the receiver they need to be more tightly coupled with the receiving thread. The separate timer thread is completely removed from PBP2. The other main difference is that. in order to correctly handle retransmitted messages. PBP2 needs to have two separate running states. This is in addition to the basic start up and shutdown states that allow group creation and correct termination.

The timer mechanism in PBP2 is implemented as part of the main thread. rather than as a separate timer thread. Timeouts are not needed to trigger resends of normal messages. They are associated with specific events not with specific messages. We implemented the

timer by using a timeout to the `select` function call that essentially makes the main thread a periodic timer. When returning from `select`, either due to a timeout or a valid read descriptor, the main process issues a `gettimeofday` call and compares the time to the various recorded timeout values. If the new time is greater than the recorded time, that timer has expired and appropriate action is taken. During periods when no timer is set, the timeout for select is greatly increased to reduce CPU contention. The benefits of this are that there is now one less thread competing for CPU cycles with the user process(es) and that this thread will consume fewer cycles during periods of message inactivity than the regular interval timer used in PBP1.

The need_resend state is important to keep a process from detecting more than one gap in sequence numbers at a time. There are three ways a process can be put in this state. The first is in the normal course of receiving messages. When a gap in the sequence of messages from a given sender is detected a NACK is sent and the process makes the transition. The second transition can occur upon the receipt of a PLAIN_ACK message that has a sequence number higher then the last message received (from the sender of the PLAIN_ACK) at this process. The third transition occurs on receipt of a shutdown message. The sequence number of a shutdown message is the same as it would be if it were a normal message. Therefore, a gap may be detected. Once in need_resend state a process will only handle NACKs and those messages it needs to fill in the sequence. Any other message could trigger another transition to need_resend. This would create bookkeeping difficulties, and significantly increase the complexity of the system. Due to the low loss rate of an Ethernet network we feel it is better to prevent a process from having more than one outstanding NACK request.

3.3 Applications

The use of efficient, reliable broadcast on a LAN can have several applications. The ability to share data within a group of processes without sending multiple messages to each member of the group can be used to implement a number of distributed applications.

3.3.1 Distributed Shared Memory

The fully-replicated model of Distributed Shared Memory(DSM) can take advantage of PBP. In such a model, each process maintains a local copy of the shared memory space. When a read is performed it is performed locally, by reading this copy. When a write is performed it is broadcast as an update to all the other processes. When an update arrives it is applied to the local copy of memory. The FIFO order provided by PBP ensures the writes are ordered by process. Using synchronization, a system can ensure a coherent view of shared copies of memory. PBP was initially designed to overcome message loss as part of the BDSM system discussed in chapter 2.

3.3.2 State Machines

Another use of PBP might be as the communication channel for a state machine [78] implementation of a distributed service. This model of fault-tolerance relies on redundant processing. Using broadcast is an efficient way to disseminate data to multiple backup processes at the same time. Since PBP will declare a process dead and reorganize itself if a given process stops participating it is ideal for fail-stop protocols.

3.4 Conclusions

In this chapter we have shown how a common reliability protocol may be multiplexed to provide FIFO ordered messages to a broadcast medium. This protocol, PBP, provides source ordered reliable message passing to a group of processes sharing an Ethernet segment. PBP provides what amounts to a series of pipelines connecting the group members. We have shown that the protocol provides two important guarantees that can be relied on when defining higher level programs. We take advantage of these properties by using PBP as the communications layer for our BDSM system. However, it can be used for other group communication applications. In chapter 4, we present the performance results of the two version of the PBP system.

```
Receiving Actions:
    1 State : Exp != resend_state
    ∧ receive m = (t, j.n.e.data) (from network or queue)

        if (i == j ) continue:
        window_base[j] = max(window_base[j]. e[i] − 1):
        if (t == PLAIN_ACK) do
            if (n ¿= exp_seqno[j])
                send m = (NACK. i.curr_base[i]. exp_seqno[1..n − 1]. (j. exp_seqno[j])):
                State : Exp = need_resend:
                nack_target = j:
                set nack timer:
            continue: // We already got ack info
        // Handle a nack message
        if (t == NACK) then
            if (data[0] == i) then
                resend message with seqno data[1]:
            else continue:
        if (n < exp_seqno[j]) then
            set ack timer:
        else if (n > exp_seqno[j]) then
            send m = (NACK. i.curr_base[i]. exp_seqno[1..n − 1]. (j. exp_seqno[j])):
            State : Exp = need_resend:
            set nack timer:
        fi
        in_buffers[j][n mod ws] = m:

    od;
```

Figure 3.8: PBP2 Normal State Receive Actions

```
Timer Event:
    5 timeout do
        if (resend timer expired) {
            send m = (PLAIN_ACK, i, exp_seqno[i] − 1, exp_seqno[0..n − 1], data):
            clear ack timer:
            reset resend timer:
            }
        if (nack timer expired){
            send m = (NACK, i, curr_base[i], exp_seqno[1..n − 1], (j, exp_seqno[j])):
            reset nack timer:
            clear ack timer:
            }
        if (ack timer expired){
            send m = (PLAIN_ACK, i, exp_seqno[i] − 1, exp_seqno[0..n − 1], NULL):
            clear ack timer:
            }
    od:
```

Figure 3.9: PBP2 Timer Event

```
Receiving Actions in resend_state:

    6 State : Exp == resend_state
    ∧ receive m = (t, j, n, e, data) (from network)

        if (i == j ) continue:
        window_base[j] = max(window_base[j], e[i] − 1):
        if (j ! = nack_target ∧ t != NACK)
            enqueue (m):
        else if (n in needed range )
            in_buffers[j][n mod ws] = m:
            if (got all resends)
                State : Exp = Normal:
                clear nack timer:
            fi
            continue: // We already got ack info
        else if ( j == nack_target ∧ n not in needed range)
            enqueue (m):
        fi
        else if (t == NACK) then
            if (data[0] == i) then
                resend message with seqno data[1]:
            fi
        fi
```

Figure 3.10: PBP2 Need Resend State

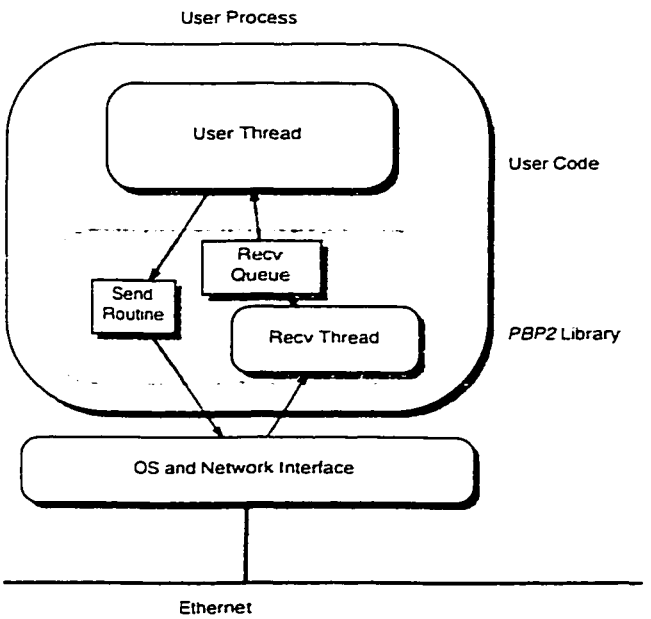


Figure 3.11: PBP2 Design Layers.

Chapter 4

PBP Experimental Results

The BDSM system is implemented on top of the PBP communication layer. The performance of application-level programs will be effected by this underlying protocol. The performance of PBP is, therefore, of interest from the perspective of BDSM. Since PBP can be used independently of BDSM as a communication layer for a different application, it is also useful to compare it to other reliable communication protocols. In order to examine the benefits of using broadcast communication, we performed several comparisons of PBP to TCP and UDP. While TCP is obviously more feature rich, it is the better comparison model because it does ensure delivery. Some tests using UDP quickly ran into lost message problems and are not shown. We also compare the performance of PBP to another reliable broadcast protocol, RMP[94], when published data are available.

4.1 Experimental Setup

Data was collected on a 20 node LAN. using P-120 PCs running the Linux 2.0.36 kernel. The network is a 10Base-T Ethernet. The systems are basically identical.The systems have common hardware as far as possible. Motherboards. network interface cards and CPUS are the same. All of the timing measurements were made using small C programs. and the `gettimeofday` system call. The timing is based on completion of the benchmark as seen by a single master process. We use the same lab setup for the BDSM results shown in chapter 6

We performed three basic timing experiments. The first is a single sender/multiple receiver setup to measure direct throughput. The time measured is for the first process to send 500 messages to each receiver and receive a single message in return from each receiver. In the case of PBP. this is done for all the receivers at the same time using broadcast. For TCP the messages are sent to each receiver and then the return messages are consumed. The second test is a multiple-sender/multiple-receiver algorithm. This is an all-to-all communication pattern where the senders and receivers are the same set of processes. Each process sends n messages to each other and awaits n messages from each other process. The time for the all-to-all experiment is measured as seen by one process. The third test is designed to be a measure of protocol overhead. by measuring latency. The lead process sends a message to each receiver (either one broadcast message or a series of point-to-point messages). It then waits for a reply from each receiver. This measures the time to get a message to each receiver and back including both protocol overhead and actual network latency.

In all of the experiments, the timing involves messages on already created channels where appropriate. We do not measure connection creation or tear-down time. Before each TCP experiment is timed, a network of connections is made. This makes the processes totally connected. For PBP, a group setup routine is called before timing experiments are started. For the experiments in which UDP was successful, while no connections are involved, all addresses and ports are resolved and bound before timing is initiated.

We use two message sizes to compare performance. Message sizes stated include all headers. Large messages are a total of 1104 bytes, while small messages are a total of 84 bytes. We feel that this is a large enough difference to ensure different behaviors. The large messages are close to the 1500 byte MTU of the Ethernet, which leads to more efficient use of the hardware. The small messages are small enough to use the hardware less efficiently. However, they are also small enough to be buffered by TCP so it is necessary to use the TCP_NODELAY protocol option to keep the system from buffering them. Since we want to account for each message on the wire we need to ensure that TCP sends a message for each send call. Under Linux 2.0 the TCP_NODELAY flag does not completely disable the Nagle algorithm. It has been shown that there is a long delay at regular intervals when using TCP with a number of small messages[64].

In most of the experiments the 95% confidence interval is under 2% of the shown time. In some of the experiments with 16 processes, despite increasing the number of samples, this interval is as much as 8% of the total time. As the number of processes increases, the number of possible delays due to processing time, interruption and message loss increases.

4.2 PBP Compared to Standard Protocols

TCP is the conventional protocol for reliable messages passing. The other data transfer protocol in the TCP/IP suite. UDP. requires application code to provide its own reliability mechanism. For a LAN environment and two or more destination processes. it can be more efficient to use a broadcast mechanism. This precludes the use of TCP because it is strictly point-to-point. In this section we look at the way PBP compares to the standard network protocols.

We compare the results of both versions of PBP using a window size of 16 to TCP and UDP. The size of the window has an effect on the throughput of the PBP protocols. A PBP process can send at most a number of messages equal to the window size before queueing outgoing messages. It then must receive acknowledgments from all the other processes to slide the window and send more messages. A larger window means more messages before this acknowledgement is required. In section 4.4 we show the impact of larger windows. When compared to TCP. the variation caused by differing the PBP window size is not readily apparent.

4.2.1 Throughput

Throughput is a measure of the amount of data that is moved in a given time. It can be obtained from a measure of the amount of time it takes to move a certain amount of data. Figures 4.1 and 4.2 show the average time it takes the timing process to deliver 500 messages to all of the other processes. In these plots a low. flatter line is closer to ideal and represents a near linear increase in throughput as the number of processes increases. For small messages,

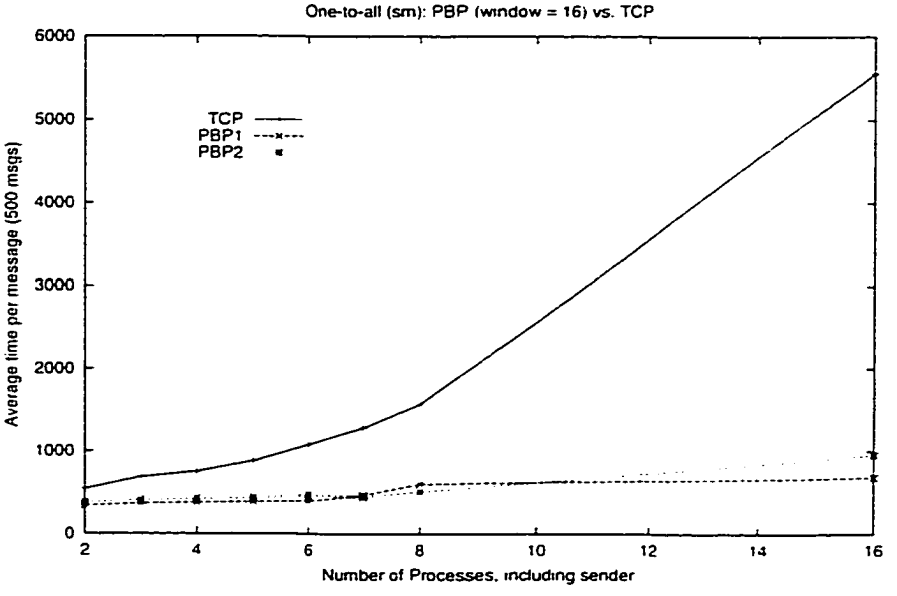


Figure 4.1: Times for Throughput Experiment for Small Messages.

TCP performs better for the 1 receiver (2 process) system. This is acceptable because it is the situation that TCP was designed for. As the number of processes increase it is clear that PBP, by taking advantage of the broadcast, is much faster. On an Ethernet, larger messages are more efficient than smaller ones. Figure 4.2 shows the one-to-all results with larger messages (1104 bytes). Here both versions of PBP are almost constant, while TCP shows a linear increase.

As mentioned above, throughput is commonly expressed in bytes per second. The 10Base-T Ethernet provides a maximum rate of 10 Mb/s. This equates to 1.25MB/s. This is the ideal maximum hardware throughput on such a network. The term *Effective Throughput* is used to describe the amount of data moved when there are multiple receivers. That is, if a process sends 1 MB of data to multiple, say 2, receivers in one second it is effectively moving 2 MB of data in that second. Table 4.3 show the effective throughput of each protocol using large messages. This table is based on the results for both PBP versions with 16 and 128

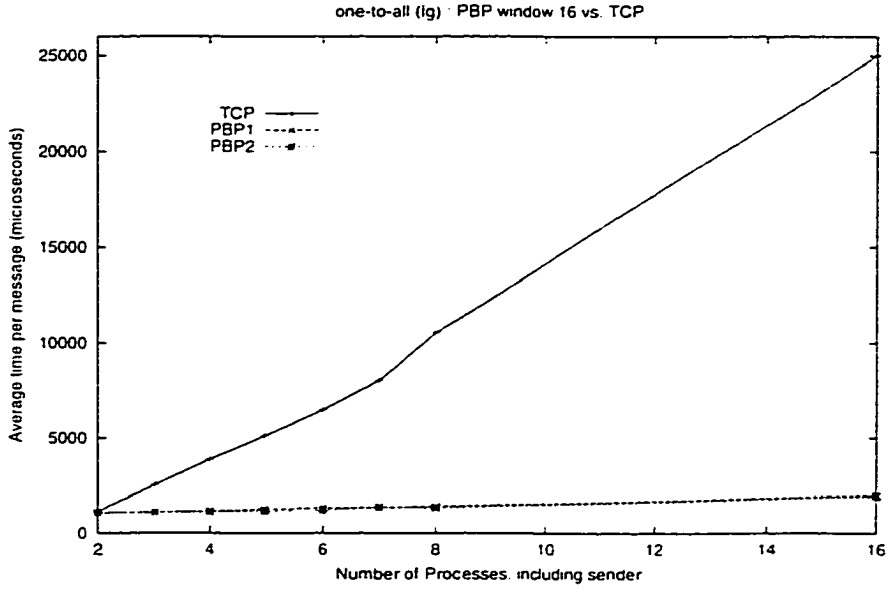


Figure 4.2: Times for Throughput Experiment for Large Messages.

Protocol	Number of Receivers			
(window)	1	4	7	15
TCP	0.918 ± 4.48 %	0.823 ± 4.33 %	0.697 ± 8.39 %	0.631 ± 8.42 %
PBP1 (16)	0.977 ± 0.28 %	3.327 ± 4.48 %	5.240 ± 2.49 %	7.924 ± 6.49 %
PBP1 (128)	0.886 ± 4.05 %	3.062 ± 0.67 %	5.173 ± 3.00 %	10.523 ± 7.15 %
PBP2 (16)	0.977 ± 0.04 %	3.653 ± 1.25 %	5.568 ± 6.03 %	8.264 ± 8.30 %
PBP2 (128)	1.105 ± 0.02 %	4.356 ± 0.87 %	7.492 ± 1.13 %	15.512 ± 2.14 %

Figure 4.3: Effective throughput in MB/s of TCP and both versions of PBP with 16 and 128 windows. Percentages are 95% confidence.

message windows. As expected, TCP shows basically a flat effective throughput because it is not sending data to multiple recipients at the same time. The PBP results show that for a large window PBP2 has a real throughput (1.1 MB/s) close to the hardware limit (1.25 MB/s). This throughput scales well, providing 15.5MB/s effective throughput to 15 processes. PBP1 is not able to scale as well. The scalability of the protocols is shown best in figure 4.4. We plot the ideal effective throughput, 1.25MB/s to each receiver compared to the window 128 PBP data. TCP is included for completeness.

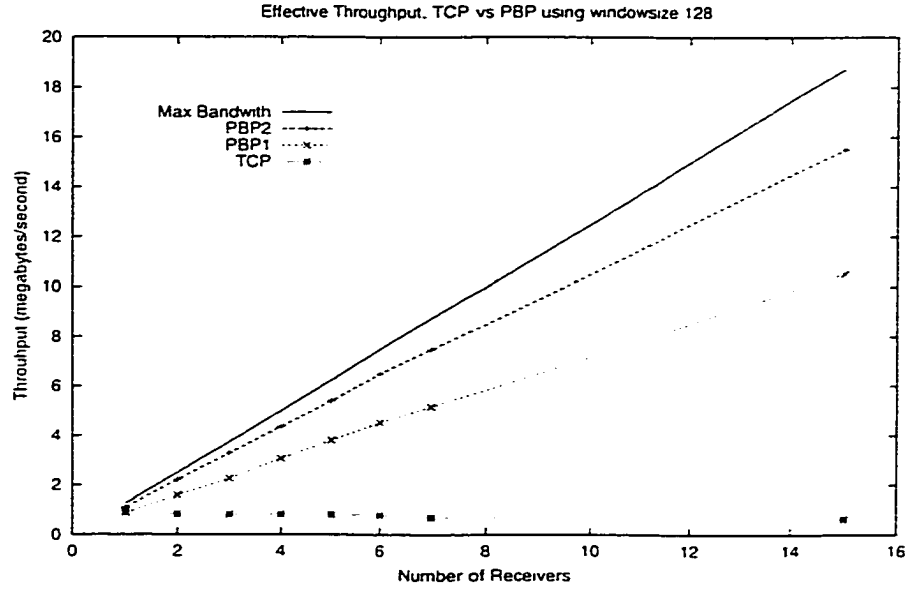


Figure 4.4: Effective throughput. Ideal versus PBP and TCP.

4.2.2 All-to-All Communication

The second set of results involves an all-to-all communication pattern. The algorithm has each process send either 500 small messages, or 50 large messages, to each other process. Here we expect to see a roughly linear increase in PBP times and a quadratic increase in TCP. This is due to the use of broadcast for PBP and the point-to-point nature of TCP connections. In a system with p processes sending n messages, the PBP system has to send pn messages. TCP, on the other hand, has to send $p^2n - pn$ messages because each message is point-to-point. Each TCP process has to send n messages to $p - 1$ other processes. The time shown is the average total time to complete the exchange as seen by the master process. The PBP system is again set to a windows size of 16.

The all-to-all performance is show in figures 4.5 and 4.6. Using small messages TCP performs slightly better than PBP for a small number of processes. It starts to get much worse for the systems of 8 processes and by 16 processes TCP is an order of magnitude

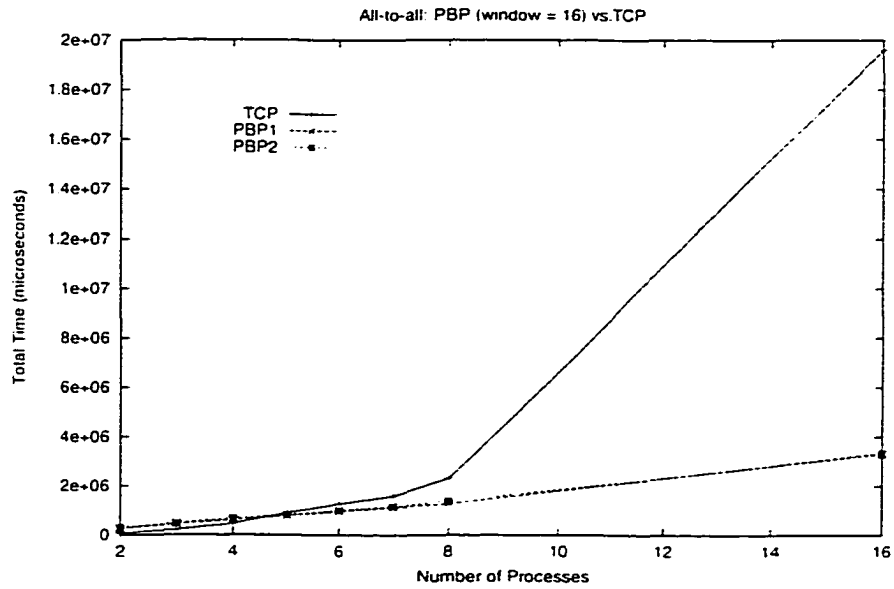


Figure 4.5: All-to-All for Small Messages.

slower than PBP. PBP shows a slow increase as processes increase. There are no steep increases. With large messages PBP performs better even for 2 processes.

4.2.3 Latency

Latency is a rough measure of round-trip time. We use it to gauge the efficiency of a protocol. Since PBP does not have as much state and overhead, it should be faster than TCP. However, it is built on top of UDP so it cannot be faster than UDP. We anticipate that the results will be in between the two. PBP also benefits from the use of broadcast. We show for small messages that TCP is actually faster than PBP2. However, when using large messages, for which PBP was designed, both versions of PBP are close to UDP in latency. PBP continues to scale better than TCP. UDP performs better because, while PBP broadcasts the initial message, it still requires $p - 1$ acknowledgements. These will have to be received and ignored by all processes except the timing process.

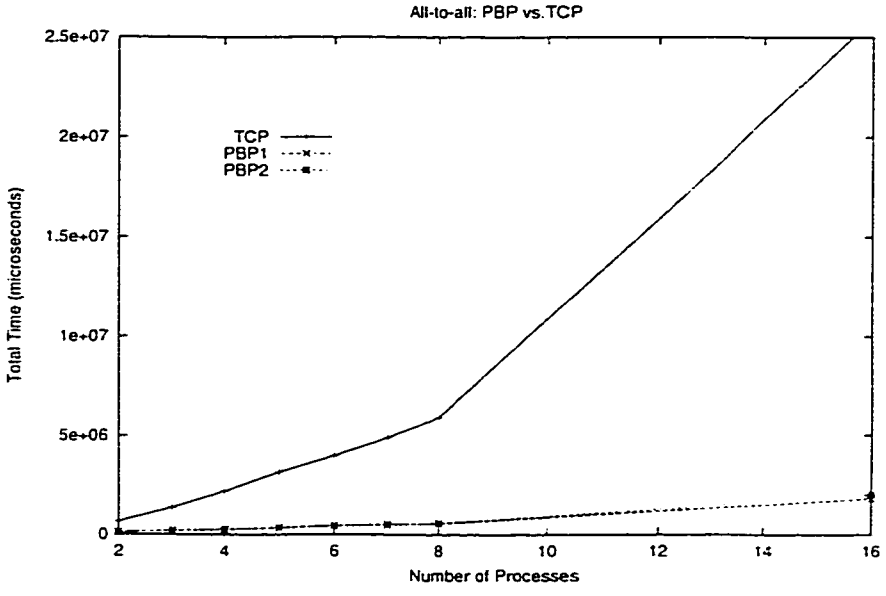


Figure 4.6: All-to-All for Large Messages.

4.3 Compared to RMP

Since TCP is not capable of taking advantage of the broadcast nature of the network it is somewhat unfair to compare it to PBP, a protocol that does. It is expected that PBP will be much faster than TCP for greater than 2 processes on a LAN. There are few published results using hardware broadcast reliably. The Reliable Multicast Protocol (RMP) is an exception[94], although the code is now commercial and is unavailable. RMP provides total order or process order for all messages. It also provides more service than PBP, as it is not limited to a single network segment. RMP is based on IP Multicast, which takes advantage of broadcast hardware when possible. RMP will perform its multicasts across network boundaries using various tree algorithms. It is important to note that these RMP results are somewhat dated. The systems used are SPARCstation2 and SPARCstation5 systems. These typically have speeds in the 50-70MHz range. These systems are slower than the P-

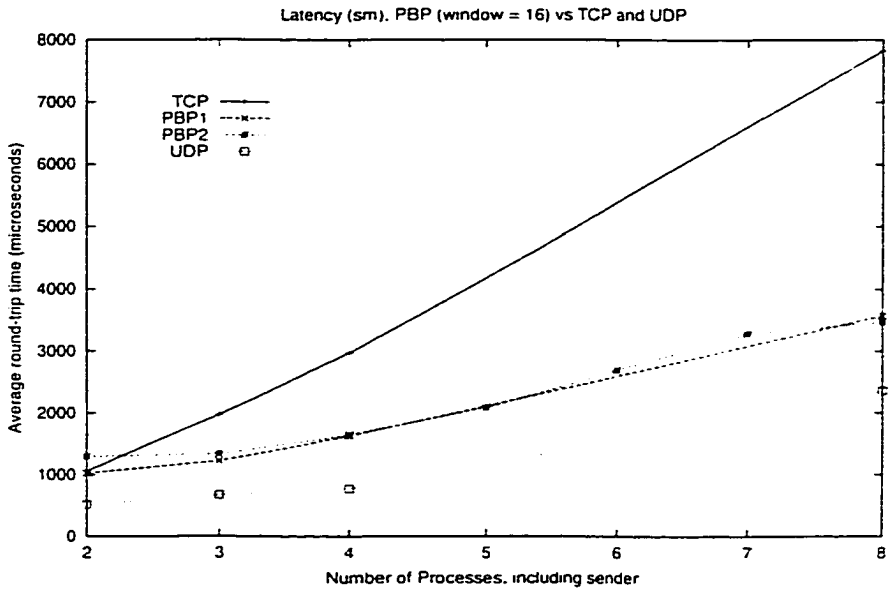


Figure 4.7: Latency for Small Messages.

120s we are using for our test-bed. However, the network in both cases is 10Base-t Ethernet. We compare both throughput and all-to-all timings to those of RMP.

RMP has published results for up to 8 receivers on a single Ethernet LAN, similar to the one used for our PBP experiments. These results show RMP with an effective throughput of approximately 4100 KB/s (4.00 MB/s) for 4 receivers and 7384KB/s (7.2MB/s) for 8. We can estimate PBP2's performance at 8.5MB/s for 8 receivers (based on the results for 7 receivers), and compare this and the results (4.3 MB/s) for 4 receivers to the RMP data. PBP2 uses the network more efficiently and provides greater throughput on a LAN. The RPM system does provides a total order while PBP2 provides FIFO order by process. This is called source ordering by the RMP authors. In a system with one sending process, total order and source order are synonymous. There is only one source, so the service provided by both systems is comparable.

Another published measure of the efficiency of RMP is the effective throughput using

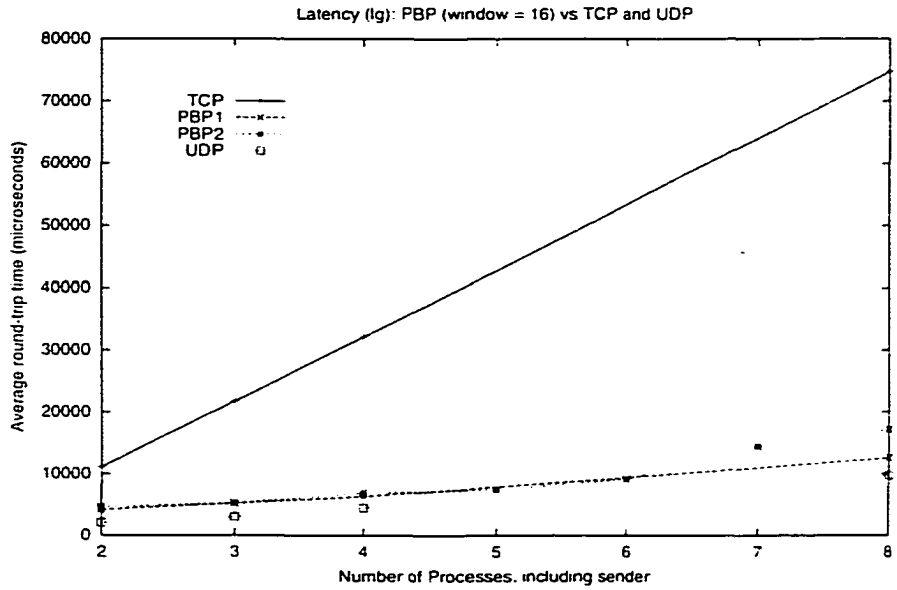


Figure 4.8: Latency for Large Messages.

8 senders and 8 receivers. This is comparable to the all-to-all experiments performed with PBP. RMP shows a throughput in this case of approximately 6000KB/s (5.8MB/s) for the 8 process all-to-all. For PBP2 with a window of 128 messages we see 6.8 MB/s. Again PBP show better performance. In this case, the fact that RMP provides total order makes a difference. All of the processes in the RMP experiments see all of the messages in the same total order. The PBP system has the possibility of processes seeing different orders within the confines of FIFO by process ordering.

4.4 Effects of Window Size

The disparity between the performance of TCP for these tests and that of PBP made using larger windows overkill. However, it is interesting to explore the effect of window size on the performance of PBP. Since PBP2 was designed to allow a larger window to increase throughput, it is useful to see how PBP performs with a larger window. We use the original

window size of 16, a medium window size of 48 and a large window of 128. We then repeated the experiments from the previous sections, excluding latency as it is unaffected by window size.

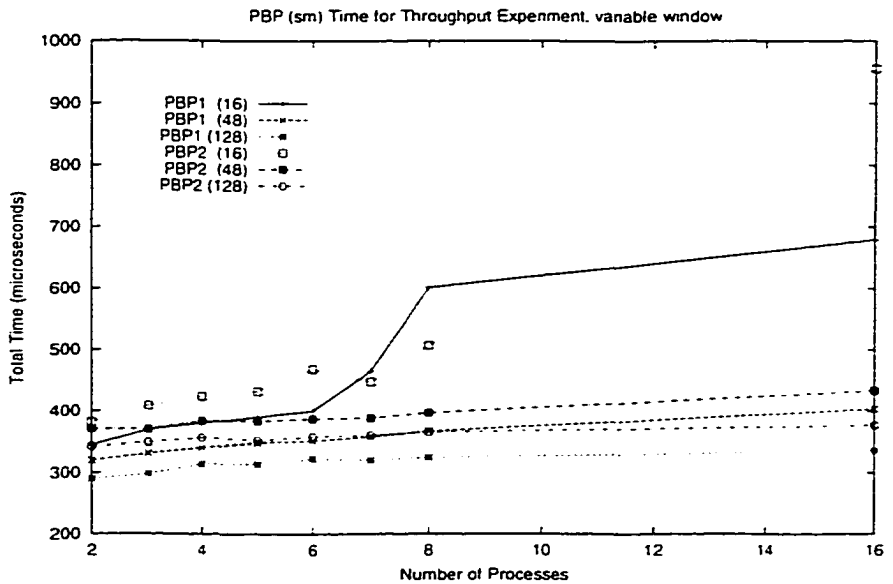


Figure 4.9: Time for Throughput Experiment. PBP with Variable Window Size. Small Messages.

Throughput for both PBP systems is much better with larger windows. A larger window means the sending process can send more messages before having to wait for acknowledgements. It can, therefore, spend more time sending messages and less time waiting for them. Figures 4.9 and 4.10 show the throughput results. In both versions of PBP, receivers track the number of messages received by any given sender. When this number reaches the window size without any piggybacked acknowledgements having been sent, a plain acknowledgement is sent. This happens without a timeout to decrease the response time. The system still experiences a delay in the sending of messages as these plain acknowledgments arrive. The larger the window, the less frequent this delay. The sudden increase at seven processes of PBP2 (48) in figure 4.10 is believed to be an artifact of the Linux kernel. See

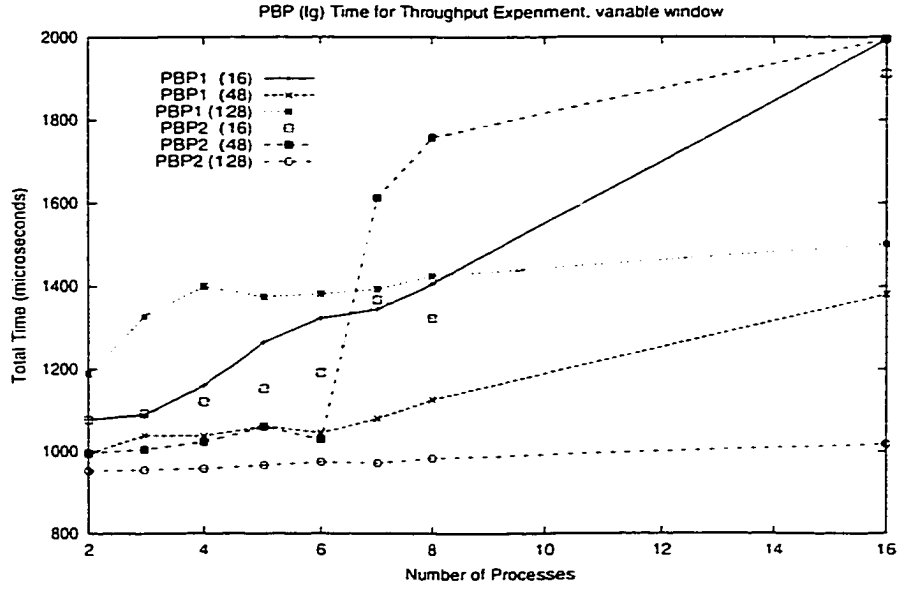


Figure 4.10: Time for Throughput Experiment. PBP with Variable Window Size. Large Messages.

section 4.5.

The large window also has an effect on the all-to-all experiments. Figures 4.11 and 4.12 show the results for the all-to-all experiments. PBP1 was designed for a small window. It would not run consistently for 16 processes for larger windows with large messages (figure 4.12). PBP2 (128) shows consistently good results, both for single sender throughput and for the all-to-all exchange.

When using larger windows, we see an increase in message loss. Figure 4.13 shows the raw data from one set of runs of the all-to-all benchmark using PBP2 with the three window sizes. The master program produces a simple count of messages it sees as lost. This is only the view of one process, but it provides an illustration of the effects of allowing more messages to be sent before requiring acknowledgement. The typical network buffer in the Linux kernel is set at 64K bytes. With a window of 48 messages we are allowing 52K bytes to be sent by each process before waiting for an acknowledgement. It is possible the

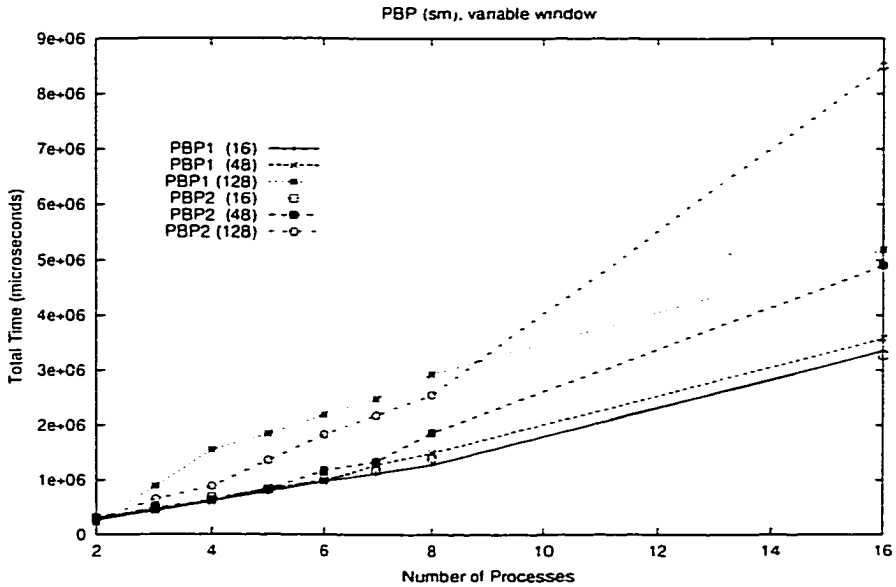


Figure 4.11: All-to-All. PBP with Variable Window Size. Small Messages.

increase in loss is due to the filling of various buffers. Another possibility may be network congestion. Messages may be dropped due to the exponential back-off algorithm. This illustrates that the number of lost messages for a window size of 16 is significantly smaller than for larger windows.

4.5 Linux Kernel Differences

The kernel version plays a roll in the effectiveness of PBP. In figure 4.10 we pointed out an unexplained, dramatic increase in time for the PBP2 benchmark with window size 48. Figure 4.14 shows a wider range of window sizes for the same anomalous execution. It shows that the PBP protocol is probably interacting badly with some part of the 2.0.36 Linux kernel implementation. To see this is so, we ran the same set of benchmarks on the 2.2 kernel. These results are shown in figure 4.15. The curves are completely different. The newer kernel version exhibits none of the peaks and valleys that appear for the mid-range

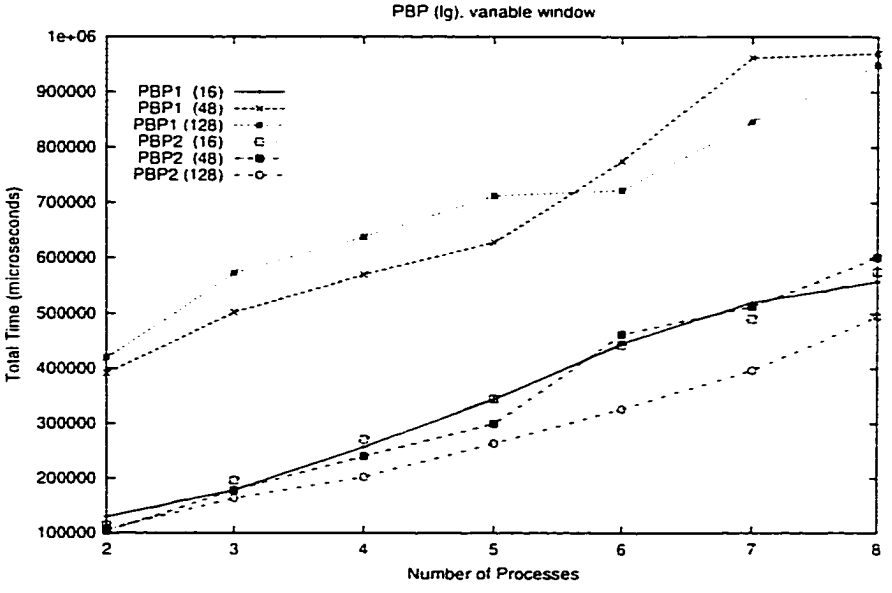


Figure 4.12: All-to-All. PBP with Variable Window Size. Large Messages.

window sizes on the earlier kernel. This difference convinced us that the strange curves in the PBP2 results were not inherent to PBP, but are an artifact of the kernel itself.

4.6 Conclusions

We have presented timing results that show that PBP provides a performance improvement over other ways to reliably send messages in a LAN environment. We did this by comparing our timing results to the industry standard point-to-point protocol for reliable message passing, TCP. We also compared our results to the published results for another reliable broadcast protocol. The above comparisons serve to show that, for application that have a flat network topology, PBP is an efficient way to make use of broadcast capabilities. Additionally, we have compared two methods of implementing the basic PBP services. These results show that the negative acknowledgement protocol performs better for the majority of uses. It justifies our use of PBP2 as the communication layer for our implementation

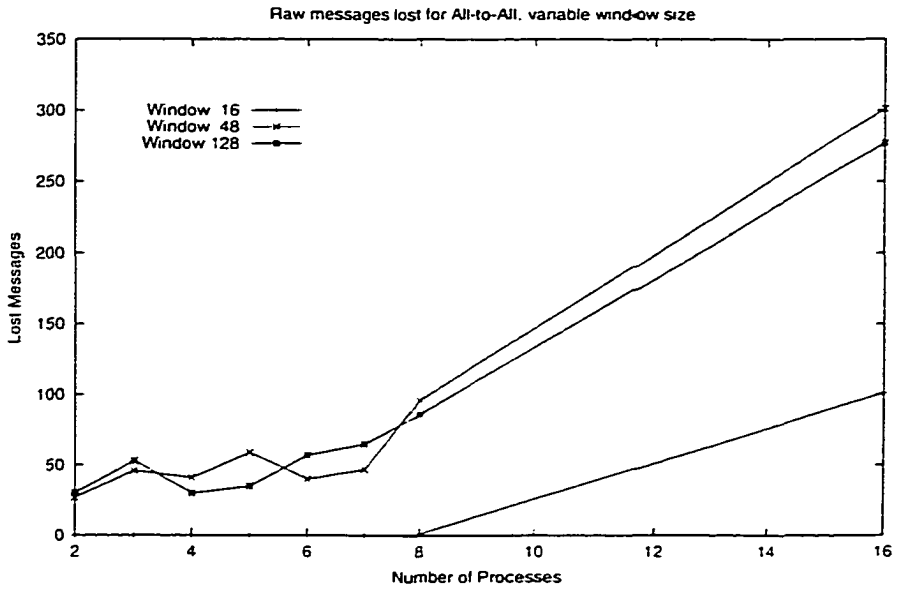


Figure 4.13: Raw Number of Lost Messages for All-to-All. PBP2 with Variable Window Size. Large Messages.

of BDSM. By using a reliable broadcast protocol for BDSM we can take advantage of the increased throughput and should see benefits, especially for all-to-all forms of data sharing, at the DSM level.

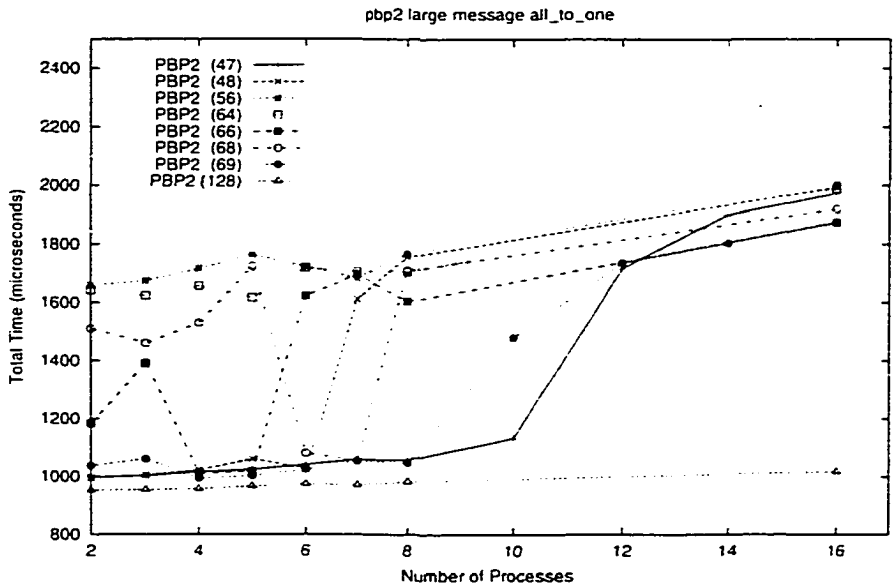


Figure 4.14: All-to-All. PBP2 with Variable Window Size. Large Messages.

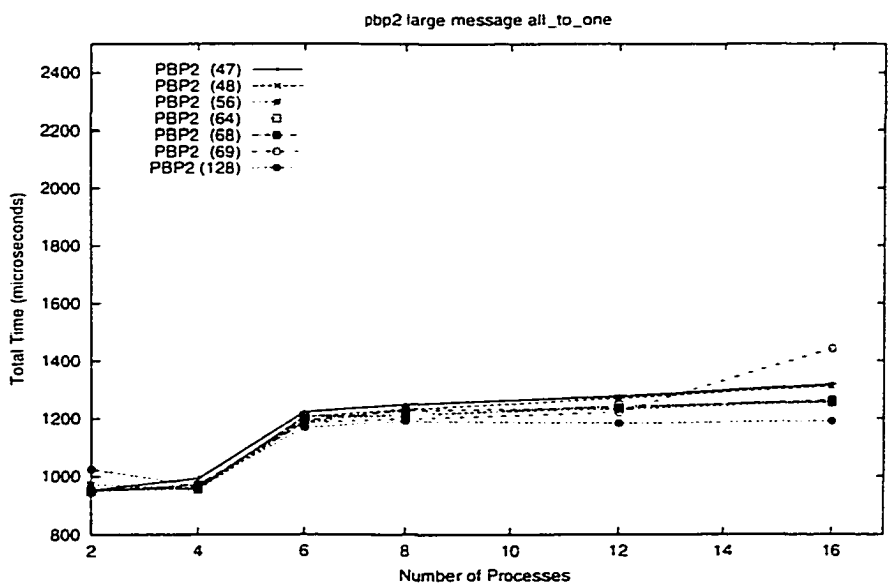


Figure 4.15: All-to-All. PBP2 with Variable Window Size. Large Messages.

Chapter 5

BDSM Implementation

In chapter 2. we presented both the design and theory behind the BDSM model and its user interface. We then presented the communication layer. PBP. that we will use under the BDSM system. in chapters 3 and 4. We now discuss the actual implementation of BDSM. We then prove that the synchronization primitives are correct and ensure BDSM coherence. Finally. we show that the implementation provides the services specified in the theoretical model.

5.1 Implementation Overview

Our system is designed for a common networking environment. We use a network of commodity workstations as a platform for the DSM system. Further. we require all of these workstations to be on the same Ethernet segment. This allows us to use *hardware* broadcast and to have a controlled message-passing environment. Each workstation will execute one user process. In turn, each user process has an associated BDSM sub-system that manages

the shared memory. There is a complete copy of shared memory on each processor. The user process can then access its copy of memory locally, with no waiting for reads or writes. Writes to memory modify the local copy and arrange to broadcast the updated values to all the other processes. We maintain the memory segment as a contiguous collection of discrete locations. Reads and writes operate at this level of granularity. The size of a location is defined by the programmer. The memory manager uses hardware broadcast to send updates to all other processors. It may buffer these updates locally to reduce the number of messages sent. Figure 5.1 shows the basic system layout. Using broadcast means that each update in an n process system is one message, rather than $n-1$ discrete point-to-point messages. Chapter 3 discusses the details the layer that actually handles this communication.

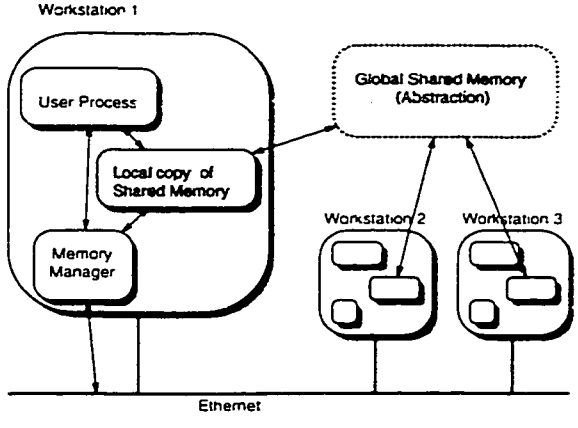


Figure 5.1: DSM system design

The distributed shared memory system is built on PBP. This provides a communication layer that both ensures delivery of all messages from non-failed processes and provides a FIFO order for all messages from each process. Each memory manager handles incoming messages from this system as they are delivered. Due to the knowledge that messages are partially ordered, incoming updates are applied immediately to the local copy of memory.

Similarly, barrier and lock messages (see below) are handled as they arrive. There is no need to reorder events at the DSM protocol layer.

In order to use PBP and the underlying Ethernet most efficiently, individual updates are buffered locally until there are enough to justify the sending of a message. Using a large message made up of a number of updates further reduces the number of messages. The number of messages that can be buffered depends on the size of the locations in a given segment. BDSM will buffer as many updates as possible for a given segment. The number of updates buffered is determined by our need to limit messages to less than 1500 bytes, the maximum transmission unit of Ethernet, to prevent message fragmentation at the IP layer and to include necessary control data.

5.1.1 Synchronization

Using message-based protocols at the PBP layer for synchronization allows us to avoid some of the pitfalls of PRAM. Synchronization under PRAM, where the actual operations are performed as PRAM memory accesses, does not provide true mutual exclusion (without a separate exclusion server[49]). For this reason PRAM, although straightforward to implement, is usually considered too difficult to program to be useful. However, in a message-passing environment there is no need to limit ourselves to using PRAM memory accesses to implement synchronization operations. Barriers and locks are implemented by message passing, the same way DSM writes are. Figure 5.2 shows the basic pseudo-code for the barrier implementation. In order to pass a barrier, a process must receive a barrier message from each other process and call `dsm_barrier` itself. All messages appear in FIFO order so for each barrier message received all previous writes by the sending process must

have been received. No process gets updates written before the barrier once it has crossed the barrier. Similarly, once a process reaches the barrier it waits until all other processes have sent it barrier messages. Therefore, even if one process gets past the barrier much earlier than the others, any writes it issues after the barrier will not be seen by any other process until it, too, has passed the barrier. No writes issued after the barrier can be seen by any process before the barrier. It is necessary to use sequence numbers on barrier messages (an alternating bit suffices) to ensure that messages are associated with the corresponding barrier. Figure 2.4 shows the barriers in use.

User calls barrier:

```
dsm_barrier(bar_num) {
    if(!barrier_started(bar_num))
       start_barrier(bar_num);
    barrier_arrival(bar_num, my_procnum);
    flush_all_write_buffers();
    broadcast_barrier_message();
    if(barrier_num_marked(bar_num)==numprocs)
       return;
    else
       cond_wait(bar_num);
}
```

Dsm thread receives barrier message:

```
handle_barrier_msg(bar_num, sender) {
   if (!barrier_started(bar_num))
  start_barrier(bar_num);
   barrier_arrival(bar_num, sender);
   if(barrier_num_marked(bar_num)==numprocs)
        cond_signal(bar_num);
}
```

Utility functions:

```
start_barrier(bar_num){ barrier_started(bar_num) = true; arrivals(bar_num) = 0;}
barrier_started(bar_num){ return(barrier_started(bar_num)); }
barrier_arrival (bar_num, process_id){ arrivals(bar_num) = arrivals(bar_num) ∪ process_id;}
barrier_num_marked (bar_num){ return(|arrivals(bar_num)|);}
```

Figure 5.2: Pseudo-code for barrier implementation

Locks are implemented as defined by Ricart and Agrawala[74]. There are two minor changes. The first is that all messages are broadcast. This means that a lock request is sent once to all processes. This reduces the number of messages per critical section from $2(n-1)$ to n. The second change is simply to require the BDSM buffers to be flushed before sending any lock protocol message. This is done exactly as with the barriers. Since there

is no point-to-point message-passing, all pending writes by all other processes will be seen by the acquiring process before it is granted the lock.

We present an outline of the lock protocol. A process, say p_i, issues a lock request when it tries to acquire a lock. This message is sent to all other processes. Then p_i blocks until it acquires the lock. Acquiring the lock means receiving a "go ahead" message from each other process. When a process, p_j, receives a request message, it is either in contention for the same lock or not. If p_j is in contention for the lock it decides, deterministically, based on the request sequence number and process id, if it should get the lock first. It then either replies to p_i or defers a reply until it releases the lock. If p_j is not in contention for the lock it replies immediately. In any case, before a process sends a lock protocol message, request or reply, it flushes all of its segment buffers.

P_0	P_1	P_2
dsm_lock_acquire(0):	dsm_lock_acquire(0)	while (read(z) != 1) skip:
$z := 1$:	read(z) = 1:	
dsm_lock_release(0);	$z := 3$:	while (read(z) != 3) skip:
	dsm_lock_release(0):	

Figure 5.3: Example using locks

In figure 5.3 we assume p_0 acquires the lock first. This is not guaranteed, but serves for this explanation. What this means is that p_0 sent a request message to p_1 and p_2. Process p_1 also sent a request to p_0 and p_2. Since p_2 is not in contention for the lock, it simply replies to any requests. So both p_0 and p_1 get a reply from p_2. The lock protocol arbitrates among contending processes in a deterministic way and we are assuming it chooses p_0 first. So p_0 defers its reply to p_1's request knowing it should get the lock first. On the other

hand, p_1 replies to p_0, granting p_0 the lock. Since no writes have been issued no updates are flushed by these messages. When p_0 releases the lock, because it deferred a reply, it sends one now to p_1. This flushes the write to z. Process p_1 now can acquire the lock, having seen all previous writes. Note that p_2 considers both acquires to have happened when it received the request messages. There is no guarantee it will escape either while loop. The write by p_0 could be overwritten by the write by p_1 before p_2 reads the 1. If p_2 needs to be sure to see the values written it needs to perform some synchronization itself.

5.1.2 Implementation Details

The BDSM system is a user-level C library which uses the Linuxthreads[61] implementation of Pthreads[12]. The library maintains a collection of DSM locations for user-level code. These locations can be of arbitrary size, up to 1276 bytes (the maximum payload of a single PBP Ethernet packet). All locations in a given DSM segment are the same size. The locations in any given segment can be read and written by location number. We also allow reads to be made directly from shared locations through pointers. This allows comparisons with and assignment *from* shared data to be transparent. However, writes are made by explicit library function calls. This is necessary to allow the write updates to be handled by the system. In this way, a segment may be treated like an array of locations. For example, "`x = dsm_segment[i]`" would assign the value of shared location i to local, unshared variable x assuming `dsm_segment` was set to the address of the base of the dsm segment. Writes must use the `dsm_write` library routine because we are not using a page-based system. So "`dsm_write(dsm_id, i, &x)`" would write the value in x into the ith location of the segment *dsm_id*. If the user makes assignments directly to the shared memory addresses

the values will not be propagated.

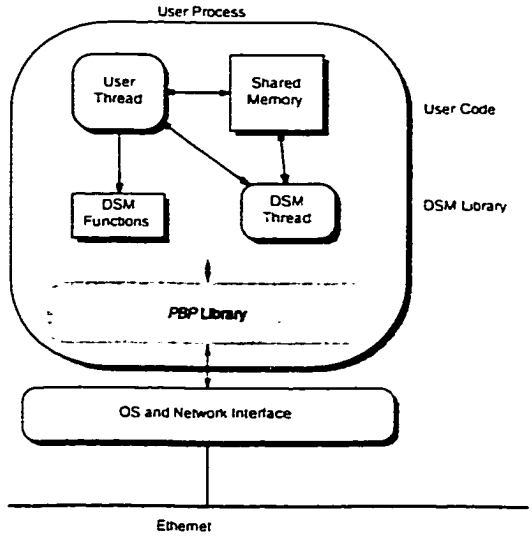


Figure 5.4: System structure

Figure 5.4 shows the structure of the complete DSM system as currently implemented. All of the threads run in the same user address space. The DSM layer consists of a single thread and a number of interface routines. The DSM thread handles incoming DSM messages by blocking on a PBP receive call. The DSM interface functions include `dsm_write`, `dsm_read`, the various synchronization operations, and the segment creation operations. The user thread communicates with the DSM interface routines and the shared memory segment.

As mentioned above, the buffering of writes is done on a per segment basis. This is primarily an implementation decision. There are two main reasons for this. The first is that, since different segments can have locations of different sizes, using a single buffer would require extra bookkeeping overhead. Secondly, it would mean associating a segment identification number with each updated value rather than with each update message. Using one buffer per segment we calculate the number of writes each buffer can hold at segment

creation time and can keep a simple count to determine when the buffer is full. When a synchronization operation is called, the buffer for each active segment descriptor is emptied in turn.

5.2 Proof of Implementation

In this section we prove that the implemented BDSM system provides an accurate realization of the BDSM model. To show the implementation is correct we will show it preserves the axioms that define BDSM. Further, we will show that the synchronization operations are correct. This means proving safety and liveness. It also means proving that the synchronization is properly ordered with respect to other operations. We then use these proofs to show that our implementation preserves the requirements of BDSM.

We make a distinction between the receipt a message and its delivery. Messages are received from the network by PBP. Messages are then delivered to the BDSM layer, in order, by way of a receive queue. A given message is not available to the BDSM system until it has been delivered to this queue.

The functionality of BDSM is based primarily on the FIFO nature of the underlying PBP layer. In chapter 3, we prove two properties about the PBP system that we rely on here. We restate them here:

PBP Property 1 (P1) : Each message sent by any process i is delivered once and only once at all processes $j \neq i$ (definition 3.2).

PBP Property 2 (P2) : Messages are delivered in the order sent. For any two distinct messages m and n, sent by p_i with sequence numbers h and k, if m is sent before n

then m is delivered before n at all j (definition 3.3).

P1 ensures the delivery of each message. Each message sent will be delivered to all other running processes, unless the sender fails. Property P2 ensures that each message sent by a given process is received and passed to the application (the BDSM system in this case) in the order it was sent by the corresponding application on the sending processor. These two properties together create essentially a set of FIFO reliable pipes among the BDSM processes.

5.2.1 Barrier Correctness

To prove the correctness of the synchronization operations we make the assumption that the program using them is correctly written. This means several things. First, the semantics of the synchronization operation are obeyed. This means that, for each barrier, all processes issue calls to the barrier routine. And, similarly, for locks, no process tries to acquire a lock which it is already holding and locks are nested but not overlapping. The second part of this assumption is that, for barriers, no calls to the same barrier identifier are adjacent in the program. Any two barriers that occur in a row have different identifiers. This assumption is natural. Programmers who don't follow the conventions of the API cannot expect correct results.

Theorem 5.1 *BDSM barriers are correct synchronization operations for correctly written programs.*

The proof consists of two elements, liveness and safety. Liveness means the program will not deadlock, with processes failing to cross a barrier. Safety is a term used to describe

the proper functioning of a barrier. Each process reaching a barrier must block until all processes have reached the same barrier.

Liveness can be shown by P1. Since we are only concerned with programs that use barriers correctly, any process that arrives at a barrier will, eventually, cross the barrier. Consider a program consisting of n processes. Each process, upon making its call to the barrier routine, will send a barrier message. And, since the program is correctly written, all process will make such a call and each will use the right barrier id. These barrier messages are guaranteed to arrive due to P1. Once a process receives $n-1$ other barrier messages it crosses the barrier and can continue execution.

Safety is ensured. No process can cross a barrier before all other processes have reached that barrier. Assume that process p_i, in an n process program, does cross some barrier b before all other processes have reached it. This means that either p_i has received $n-1$ barrier messages for b, one from each $p_j, j \neq i$ or it has crossed the barrier having received less than $n-1$ other messages. In the first case, since P1 ensures "only once" delivery, some process must have sent a barrier message for b without reaching b. This would be a violation of the protocol. Since the user program is correct, it is not possible for this to be an old message for a different instance of barrier b because there must have been a barrier b' since the last use of b. In order for b' to have been crossed, thus allowing b to be used again, all of the previous messages for b must have been consumed. For the second case, process p_i must have violated the protocol to cross a barrier with less then $n-1$ barrier messages. This would require a Byzantine failure mode we are not concerned with. Since it is impossible for any p_i to cross b without all the other processes arriving at b, safety is assured.

Theorem 5.2 *BDSM barriers correctly preserve the BDSM order requirements with respect to updates.*

All writes made by process p_i before reaching a barrier are sent before the barrier message is sent. Since each other process must get the barrier message before proceeding, all of these earlier writes must be received as well, due to P1. This means all other processes must get all of the writes issued by p_i before getting the barrier message from p_i. And since each process behaves this way all writes before the barrier are seen by all processes before the barrier is crossed.

Conversely, because a process blocks until a barrier is crossed, any writes that are sent by a process after the barrier will not be seen until this process has also crossed the barrier. Consider p_i and p_j. If p_i sees an update before barrier b issued by p_j after b then p_j must have crossed b. In order for that to have occurred, p_j must have received a barrier message for b from p_i. However, since p_i has not reached b yet (it is reading values before b), such a message has not been sent. Therefore, p_j cannot have crossed the barrier and p_i cannot have received an update from p_j written after p_j crossed b.

5.2.2 Lock Correctness

Theorem 5.3 *BDSM locks are correct synchronization operations for correctly written programs.*

Ricart and Agrawala proved both liveness and safety for the distributed mutual exclusion algorithm they designed[74]. Our locks differ only in that they use broadcast rather than point-to-point messages and that there is buffer flushing done when messages are sent.

Neither change effects the validity of the original proofs. Using broadcast messages simply means some processes get messages they don't need. These are ignored. Sending update messages due to buffer flushing has no bearing on the mutual exclusion protocol. Therefore, we conclude that our lock implementation satisfies the safety and liveness requirements as well.

Theorem 5.4 *BDSM locks correctly preserve the BDSM order requirements with respect to updates.*

Proving the ordering requirements are met is more complicated. Locks are essentially global communication similar to barrier, except each process does not block. To see how this is so, consider that each process must receive a request and reply to it. The receipt of this message and the reply mark the time in the receiving process' view that the lock was acquired. And, since we are using broadcast, all processes in the system will receive a reply message. Processes that are not in contention for the lock will not wait for the reply. However, the updates that are flushed by the reply will still be applied at all processes.

Step one is to show that all previous writes are seen before a lock is acquired. Since a process is required to flush its buffers before sending a lock message all previous writes will be sent first. Property P2 provides for the order of these sends to be preserved at all receivers. Because the acquiring process must receive a message from each other process it must receive all earlier writes from each other process as well. Therefore, in order to acquire a lock all previous writes must be seen.

The second step is to show that no writes after a lock is acquired by one process are seen by another before the lock is seen to be acquired. As mentioned above, the notion of

a non-contending processes seeing another process acquire a lock is when it replies to that acquire request. Once a process issues a request for a lock it blocks. It may not execute any reads or writes until it acquires the lock. To acquire the lock it must receive a reply from all other processes. Therefore, all other processes must have replied to the request (and seen the acquire) before the requesting process can issue any writes.

5.2.3 BDSM Implementation Correctness

We now prove that the implementation of BDSM using PBP is a correct representation of the BDSM model. This is done by showing that all of the ordering requirements for the model a preserved in the implementation. Using theorems 5.2 and 5.4 and the PBP properties P1 and P2 we show that the ordering requirements are met.

Theorem 5.5 *The implementation of BDSM correctly realizes the BDSM model.*

To prove this we will show that each axiom from section 2.2.2.3 is preserved by the implementation.

1. Axiom 2.1: *Locally, all events are in program order.*

   This is preserved because processes execute in program order and writes are applied immediately to the local copy of memory.

2. Axiom 2.2: *Write leads to updates, and a write comes before its updates.*

   When a process issues a write this information will be sent as an update, either when a buffer is full, when a synchronization operation requires it, or immediately if buffering is disabled. Additionally, since the write triggers the updates, the write must come before the updates, from a global perspective.

3. Axiom 2.3: *Updates for writes to the same segment by the same process are seen in the order written.*

   Writes to the locations of each segment are buffered in the order issued. These buffered writes are then sent as updates. On receipt of an update message, a process will apply these individual writes in the order they appear in the update message, which is the order buffered. PBP property $P2$ ensures that these updates arrive in the order sent.

4. Axiom 2.4: *Barriers are in all processes.*

   This is semantically required. If it doesn't hold, the program (not the implementation) is incorrect. A process must receive a corresponding barrier message from each other process. Failure to do so indefinitely blocks the process.

5. Axiom 2.5: *Barriers are totally ordered, and all processes see the same order.*

   On arrival at a barrier, a process sends a barrier message to all other processes. It then waits for a similar message from each other process. A process performs no local actions until the barrier is crossed. Therefore only one barrier may be active at a time. Each process must cross that barrier before arriving at another. Theorem 5.1 shows that the implementation correctly preserves barrier semantics.

6. Axiom 2.6: *Updates for writes before a barrier are seen by all processes before the barrier.*

   This follows from theorem 5.2.

7. Axiom 2.7: *Updates after barrier seen after.*

   This follows from theorem 5.2.

8. Axiom 2.8: *Lock acquires are seen by all other processes.*

   In order to acquire a lock, p_i must receive permission from each other process, in the form of a lock message. The sending of a permission message in reply to a lock_request corresponds to the acquire event in the permission granting process. From the lock definition there must be such an event in each process or the lock cannot be acquired.

9. Axiom 2.9: *There must be a release for each lock acquired.*

   This is semantically required. A program that fails this is incorrect and deadlock prone.

10. Axiom 2.10: *Lock acquires are ordered, and a lock-holder's release comes before the next acquire.*

    This holds due to the implementation of locks, theorem 5.3.

11. Axiom 2.10: *Earlier updates by other processes must be seen before acquiring a lock.*

    This holds due to the implementation of locks, theorem 5.4.

Since the axioms that define BDSM are all preserved by our implementation, the implementation correctly provides BDSM coherence. We have shown that the implementation of BDSM, using FIFO broadcast provided by PBP, is a correct realization of the BDSM coherence model.

5.3 Conclusions

We use the PBP communication system presented in chapter 3 as the basis for an implementation of a BDSM system. In this chapter we discussed the implementation of this

system. Further, we have shown that our system, as implemented, provides an accurate realization of the BDSM model. In the next chapter, we provide a suite of test applications using BDSM and explore their performance on BDSM and MPI.

Chapter 6

DSM Experimental Results

In preceding chapters we presented the theoretical model and implementation details of the weak, broadcast DSM system, BDSM. In this chapter, we look at some of the performance results we have obtained using this system. We have developed a suite of test programs on BDSM. We discuss these programs and their communication patterns and compare them to a message passing alternative on the same hardware setup.

To date, we have focused primarily on parallel, numerical calculations where pure performance gains are desired. We have developed a small test suite of programs, loosely based on the SPLASH-2[80] suite. Since our primary concern is the operation of the DSM system and not the overall performance of our test programs, we have used straightforward, often naive, parallel algorithms. We present comparisons between our system and a message-passing system. We have chosen to use MPI[39] because it is commonly used for parallel programs on networks of workstations. We compare our execution times to those of similar programs using the mpich (v1.1.1)[45] implementation of MPI on the same network. We also explore some of the ways in which using PBP effects the execution of programs on

our system. One of the main benefits BDSM has over MPI is that BDSM uses broadcast communication for its collective communication operations, which essentially they all are. MPICH uses tree algorithms on top of TCP to perform collective communication operations. Recent work by Chen, Carrasco and Apon [31] attempts to implement these MPI operations using IP Multicast.

6.1 Experimental Setup

Experiments are performed on a single Ethernet subnet. The lab we use is a public access teaching lab. We do not have exclusive access to the systems. Therefore, the programs have been run in a non-controlled environment. Some of the uncontrollable factors include users logging in, NFS activity, cron jobs, and system daemons. Because of these potential outside influences we have made every effort to run our tests late at night and very early in the morning. The lab consists of up to 20 Pentium 120MHz systems. These systems run identical installations of Linux, using the 2.0.36 kernel. There is a shared NFS system where binaries and initialization data reside. Results are stored locally to avoid using NFS during the actual computations.

We replicated our speedup experiments until a reasonably narrow 95% confidence interval was obtained. In presenting our speedup results, we simply plot the mean of the replicated experiments. The 95% confidence interval is consistently no more than 2% of the plotted value.

Since the performance of PBP2 was shown, in chapter 4, to be better than PBP1 in most cases, PBP2 was used for the BDSM experiments. Additionally, PBP2 should be less CPU

intensive since it removes the interval timer thread and can increase its timeout when the communication channel is idle. The results were obtained using a window size of 16. This is the default for both versions of PBP. We also performed the experiments using windows of 16, 48 and 128 messages, as with the throughput experiments in chapter 4. The results for all of these are shown in section 6.3. While it is clear from the throughput experiments in chapter 4 that windows size can effect performance, this effect appears to be minimal for the larger programs at the BDSM level.

6.2 Test suite programs

To explore the performance of the BDSM system we developed a suite of five common parallel programs. Our suite consists of:

- `matmult`: matrix multiplication
- `nbody`: N-body particle simulation
- `jacobi`: Jacobi linear equation solver
- `cg`: conjugate gradient
- `tsp`: traveling salesman problem

We chose not to use the common benchmarks of the SPLASH2 [80] suite because these programs are designed for completely transparent memory systems. While it would be possible to port these programs to our system, the programs would be different enough that the results would not be comparable to other DSM systems. They would not serve as a true benchmark.

We use a single process version of each program as the basis of our speedup measurements. We also developed MPI versions of each algorithm. We attempted to make the DSM and MPI programs as similar as possible to ensure there were few algorithmic differences. When possible we make sure that data are initialized with the same values and that the actual computations are identical. To some extent, differences are unavoidable because the sharing patterns of DSM and message-passing algorithms are generally different. For example, we did not force the MPI code to use collective communication operations at all times, which the BDSM system, effectively, does. Our goal is to show that the BDSM system's performance is comparable to message passing on the same hardware, not that it is better.

The MPICH implementation's `ch_p4` device is used on a network of workstations. The underlying communication is done using TCP. Collective communication operations are made up of individual point-to-point messages. The algorithm used depends on the operation. For example, an MPI broadcast uses a binary tree algorithm and an MPI all-gather uses a rotation algorithm. Barriers are done using a ring and a token. The token passes to each node twice, the first to signify arrival at the barrier and the second time, departure. Our BDSM barriers consist of a single round of messages. Additionally, BDSM uses actual broadcast for all message traffic, so we expect it to perform better for algorithms that use primarily collective communication operations.

6.3 Results

The first program in the test suite is a square ($m \times m$) matrix multiplication program (`matmult`). The code uses statically initialized operands. We are using a naive row par-

titioning algorithm for simplicity. Each process is responsible for m/n rows of the result matrix. After completion, a designated process reports the result matrix. The MPI program is similar, with a designated master process collecting each other process' result rows and reporting the result. The master is also a worker so the number of compute elements is the same for both MPI and BDSM versions. The communication pattern is, effectively, a single round of all-to-one message passing in both cases. For verification, each version of `matmult` can compute the result matrix at the master process and compare the results reported by the group computation.

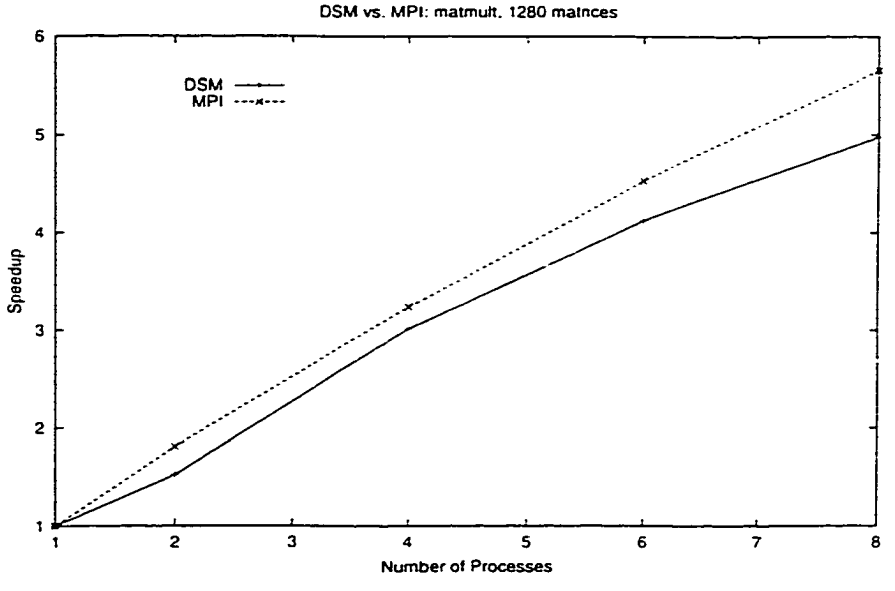


Figure 6.1: Speedups for `matmult`

Figure 6.1 shows the comparative speedup of the two programs for dense 1280 × 1280 integer matrices. Both programs exhibit similar speedups. MPI performs slightly better than BDSM, primarily due to the fact that our DSM always uses all-to-all communication. Processes that don't need the results still must handle all of the messages. The all-to-one communication pattern is essentially the opposite of broadcast. In a sense, it represents a

worst case communication pattern for our DSM system.

The second program is an N-body particle simulation (nbody). We calculate the forces and new positions of p particles in 3-space. Initial positions and masses are generated, and a single large particle is placed centrally in the space. We us a simple $O(p^2)$ algorithm where each of n processes is responsible for calculating the forces on its p/n particles by all p particles. In each calculation phase the new positions are computed and data is exchanged among all process. N-body exhibits an all-to-all communication pattern with a significant amount of computation between each communication round. The original algorithm is from a Fortran MPI implementation by David Walker[93]. We modified one copy to use BDSM and the other so the communication patterns are more similar. Initially, the MPI version used a circular loop of processes and $n/2$ communications per time-step. Our versions, both BDSM and MPI, use n rounds per time-step where each process computes the forces on its particles by each other process' particles.

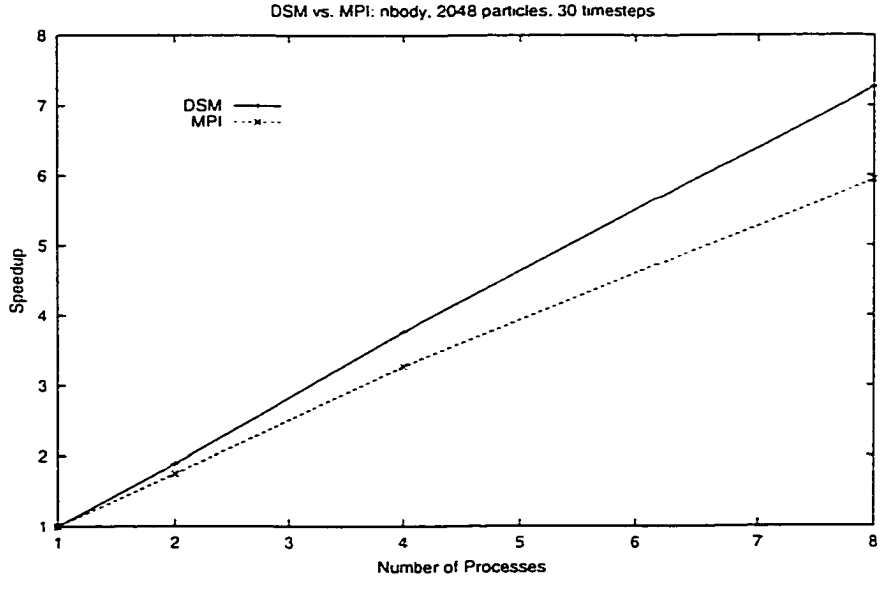


Figure 6.2: Speedups for nbody

The results shown in figure 6.2 are for 30 time steps of 2048 particles. Here, the benefits of all-to-all communication using broadcast favor BDSM over MPI. Both programs exhibit nearly linear speedup. The program will transmit a large amount of data per time-step. However, it uses relatively few time-step iterations. In the MPI version, each process swaps its particles around a ring with its neighbors. It then computes the forces of the newly received particles on its local particles. This takes place in a series of one-to-one communications, while the BDSM version performs a larger all-to-all exchange once per time-step. The amount of data that needs to be moved is the same. The same pseudo-random number seed it used to generate the random particles for each execution of each version so the computations are the same.

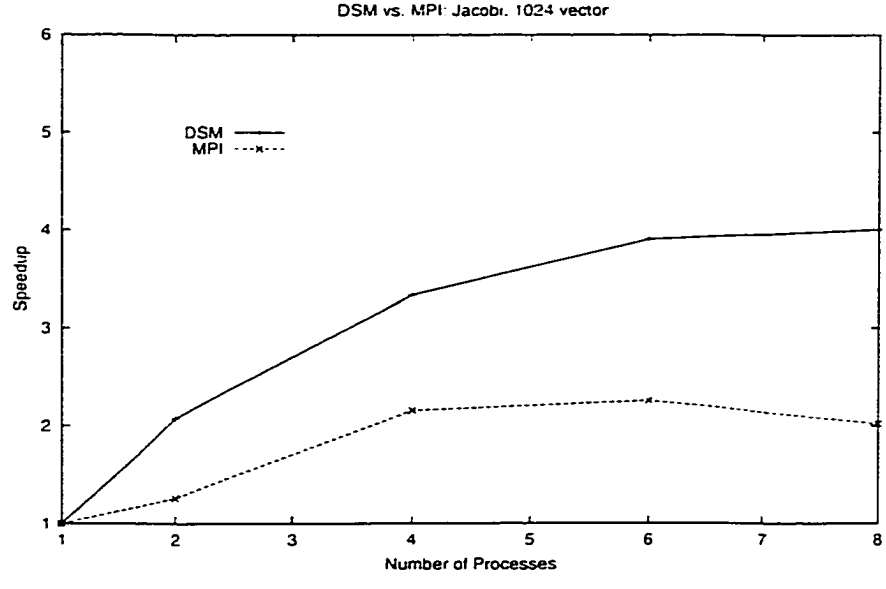


Figure 6.3: Speedups for `jacobi`

The third program is a Jacobi linear equation solver (`jacobi`). It uses an iterative approach to solve for x in the system $Ax = b$. Input data for b is generated randomly by a designated process. Matrix A is a fixed 5-diagonal $m \times m$ matrix. We use a seed that

generates a data set that converges in approximately 14000 iterations. The BDSM program was developed locally, based on pseudo-code in a causal memory paper by Ahamad, Hutto and John[5]. The MPI version was based on the one in the Pacheco's book [69], modified to make it work the same way as the BDSM version. The communication pattern is a series of all-to-all data exchanges as each process computes m/n vector elements during each iteration using the entire vector from the previous iteration. Figure 6.3 shows the resulting speedup for a 1024 element solution vector. This program consists of a relatively small amount of computation for each iteration so the efficiency of the BDSM collective communication operations is seen.

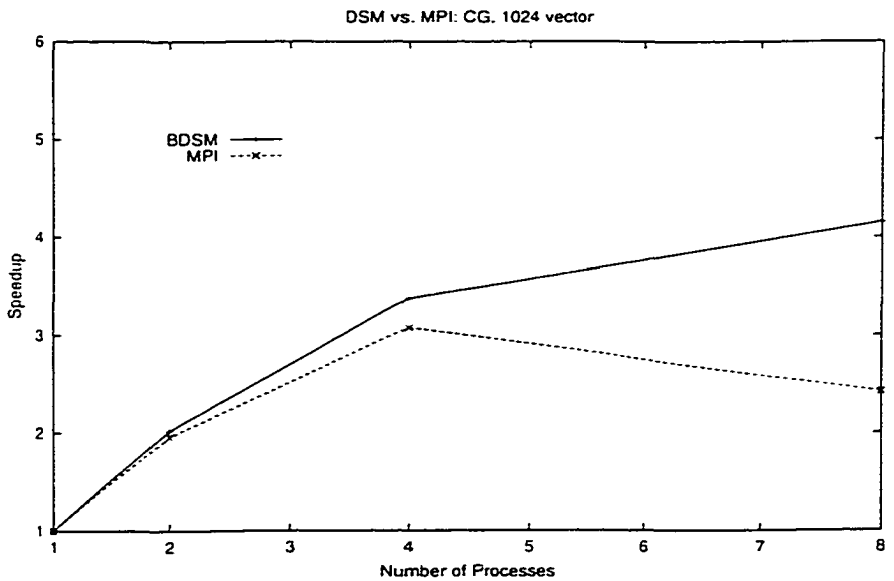


Figure 6.4: Speedups for cg

The conjugate gradient program (cg) is similar in structure to `jacobi` except there are three all-to-all exchanges for each iteration. We use the same initialization technique as in `jacobi`. Executions are based on 10000 iterations. Both the DSM and MPI programs were derived from the respective `jacobi` versions. The DSM version uses a number of different

segments for problem and temporary data. This contrasts with `jacobi`, which uses two, one for A and one for both x and b. This is the most network-intensive of the programs in our suite, and it is used to explore the message loss patterns. Figure 6.4 shows the speedup results for a 1024 element vector. MPI performs nearly as well as BDSM until the benefits of broadcast all-to-all communication dominate. Both `cg` and `jacobi` scale poorly. The sheer number and cost of all-to-all communications, even with BDSM using hardware broadcast, outweigh the benefits of more computational power for this problem size.

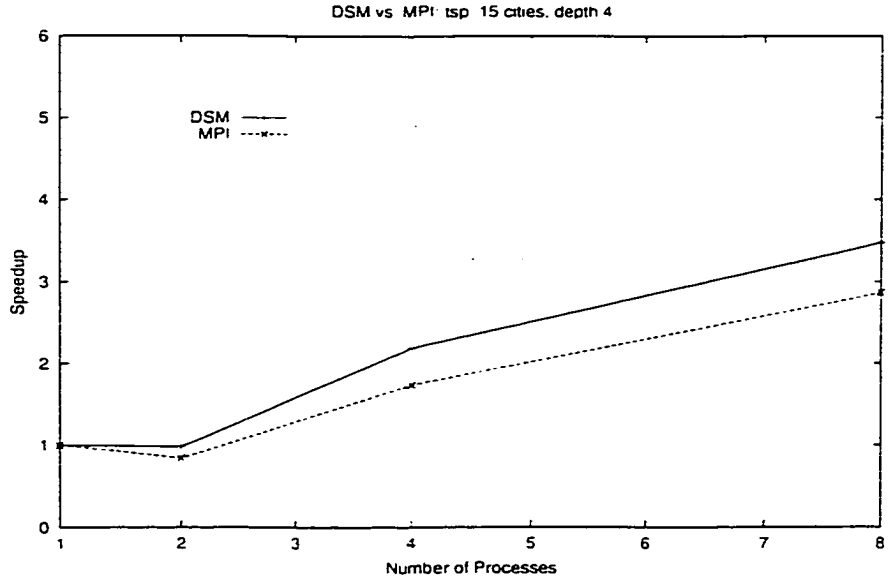


Figure 6.5: Speedups for `tsp`

The final program in the suite is a simple traveling salesman problem solver (`tsp`). The program uses a master/slave structure where a designated master process creates initial, four-city-deep paths and distributes them to the other $n-1$ processes. The master gives a job to each process in strict rotation, so the work is not necessarily evenly balanced if some paths can be dropped sooner than others. In order to keep the MPI version from having an

effective barrier, in the form of an MPI_Allreduce, we only track the shortest current path at each slave. This means the slaves do more work than is needed. This can be an advantage of the shared memory model. It would not require more communication to allow the DSM slaves to see each other's current minimum. The distance matrix for `tsp` is generated randomly. Figure 6.5 shows the speedups for a 15-city tour. The results are poor for both MPI and BDSM. This is mainly due to the inefficiency of the algorithm we are using. The single process version will not perform extra work since it keeps a global minimum path and can drop infeasible paths sooner. Also, as expected with the master/slave structure, using only one slave is actually slower than the single-process version. The communication pattern for `tsp` is basically one-to-one as the master passes each slave a task to work on in turn. It would be possible for the BDSM version to take advantage of global knowledge and improve its performance. This is another example of a program for which broadcast communication is not necessarily ideal. The code for `tsp` was derived locally, based on examples seen in course work.

Figure 6.6 shows a comparison of the number of messages sent and the amount of data transfered by each application for different system sizes. The number of messages shown includes only sequenced messages, both data updates and synchronization messages. It doesn't count retransmissions and non-piggybacked acknowledgments sent by the PBP layer. Similarly, the total bytes sent counts the number of bytes in the same subset of messages. We can see that the iterative programs, `cg` and `jacobi`, transmit large numbers of messages as each iteration involves at least one all-to-all communication. The `matmult` program, due to the size of the data set, requires a large number of messages for its single communication round. Additionally, these messages are being received and handled by all

processes. even though only the first process cares about the results. The `nbody` program. since it only runs 30 time steps. has a smaller number of messages than the other two repetitive programs. However. the amount of data moved per iteration is much greater. Each particle is represented by seven `double` variables. Running for 10.000 time-steps would transfer on the order of 1.5 GB. The simple `tsp` program creates few messages. resulting in a small amount of data transmission.

	Procs	Messages	Bytes
CG	2	153308	63527492
	4	253320	75448816
	8	493348	104091936
Jacobi	2	133303	61038732
	4	213319	70480528
	8	413355	94164608
Matmult	2	10249	14296012
	4	10257	14296868
	8	10273	14298580
N-Body	2	5862	4482104
	4	5988	4497368
	8	6360	4541816
TSP	2	88	57920
	4	104	60248
	8	134	64672

Figure 6.6: Message passing for DSM programs

6.4 Effects of Window Size

In chapter 4 we've seen marked differences in the performance of PBP2 for different. specifically larger. windows. The previous results all use the same 16 message window. In this section we show a comparison for a few test programs of 48 and 128 message windows as well. Our results show that for some experiments the larger window make little difference.

For others, larger windows can actually be detrimental.

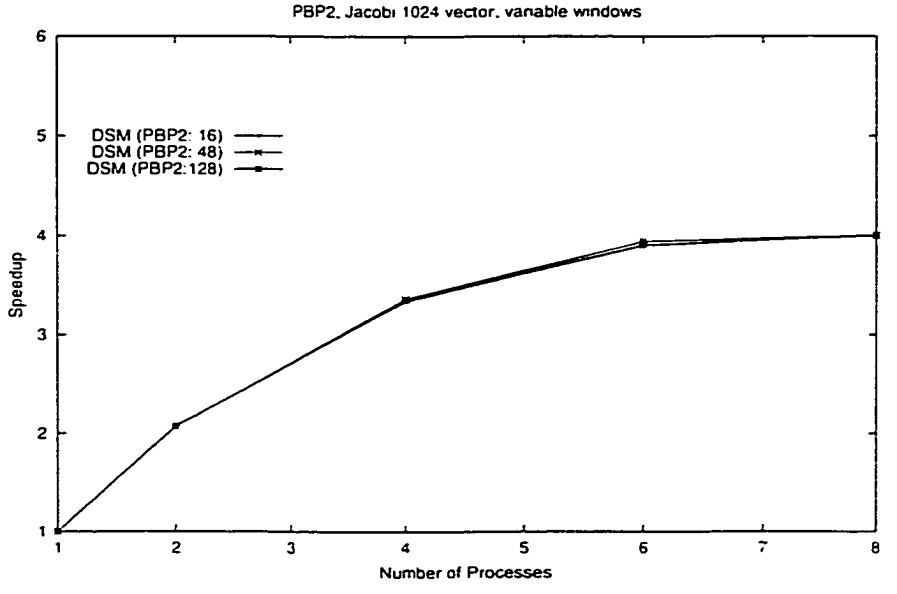


Figure 6.7: Window Size and Speedup for `jacobi`

Figure 6.7 shows that the difference in window size has little effect on `jacobi`. The results are the same for `cg` . The different window sizes produce almost identical speedups. At each iteration, each process sends less than 16 messages so a larger window should have no effect. The results are different for the programs that have more data movement and less synchronization. Figure 6.8 shows comparison for matmult. In this program, all of the processes are sending large amounts of data at roughly the same time. They are sending more than 16 messages, so the larger windows are allowing more messages to be sent at once. This increases the contention on the network. Increasing the number of messages also increases the contention for buffer space on both sender and receiver. This may, in turn, increase the message loss rate.

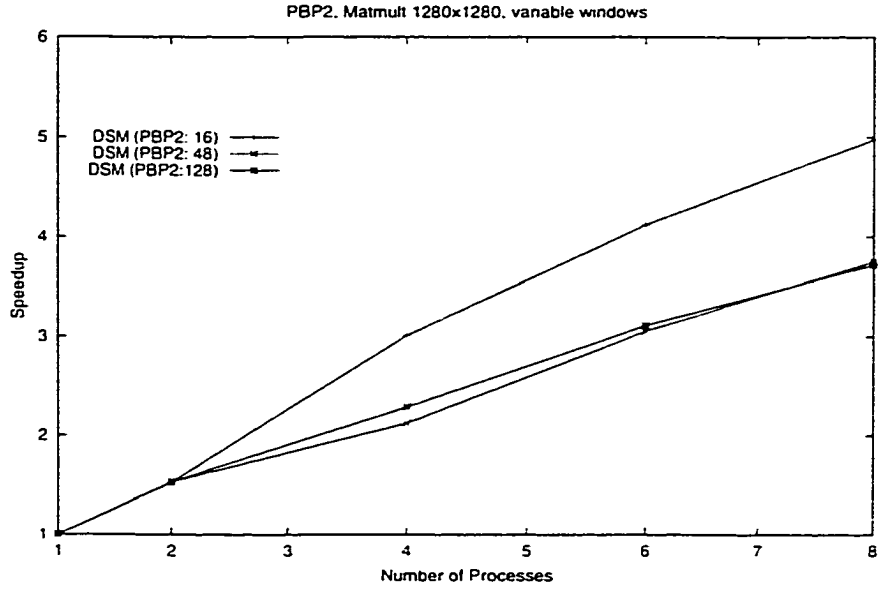


Figure 6.8: Window Size and Speedup for `matmult`

6.5 Message Loss Behavior

In this section, we explore the behavior of the underlying PBP layer and the Ethernet itself when used for the DSM computations in our test suite. Our focus will be on `jacobi` and `cg`. The other three programs exhibited no appreciable message loss due to their relatively light communication needs. The data we present here is meant to illustrate the message loss rates and provide us with insight into why messages might be lost. However, the very act of collecting accurate message loss data perturbs the network and the computation. It creates its own extra-computational messages. It also increases the computation times. This may serve to reduce the number of messages lost, by slowing the program, thereby reducing network contention. It may, however, increase the number of lost messages by adding to the network load due to the message passing required for retransmission. Since the timing numbers are not precise due to this perturbation, we only repeated the experiments a small

number of times. This leads to a large variance. The results in this section should be seen as illustrative rather than definitive.

PBP1				
	Procs	Sent	Lost (95% conf interval)	Rate
Jacobi	4	213319	15.50 ± 16.91	0.00727%
	8	413355	55.50 ± 21.37	0.01343%
CG	4	253320	25.50 ± 26.39	0.01007%
	8	493348	133.250 ± 26.13	0.02701%
PBP2				
	Procs	Sent	Lost (95% conf interval)	Rate
Jacobi	4	213319	16.00 ± 14.90	0.00750%
	8	413355	51.66 ± 27.70	0.01250%
CG	4	253320	27.33 ± 18.65	0.01079%
	8	493348	127.00 ± 26.29	0.02574%

Figure 6.9: Message loss for cg and jacobi

We begin with the actual message loss rate. Figure 6.9 shows the rate of message loss. for cg and jacobi running with 4 and 8 processes for both PBP1 and PBP2. The two-process versions of these programs lost. on average. less than one message per run and are. therefore. not very interesting. In the worst case. cg with eight processes. 99.97% of all messages were successfully delivered without retransmission. As more processes are added we see a rise in message loss. This is most likely due to the fact that with more processes there is more competition for receive buffer space and more contention on the network. A two-fold increase in the number of processes. while creating a proportional increase in message traffic, causes a 400% rise in lost messages for cg and a 250% rise for jacobi. One of the potential causes may be the interference of other processes on the system. As mentioned earlier. we do not have access to a completely isolated network. With more BDSM processes it is more likely that there is competing activity on one of the

workstations. The large confidence interval for these loss numbers, see figure 6.10, may be partly due to interference from computing activity outside of our experiments. However, it is most likely due to the low number of repetitions.

PBP1				
	Procs	Total Lost	Lost By 1 Receiver	Lost at Sender
Jacobi	4	15.50 ± 16.91	1.75 ± 1.52	13.75 ± 17.83
	8	55.50 ± 21.37	14.25 ± 5.72	41.25 ± 19.91
CG	4	25.50 ± 26.39	5.75 ± 2.39	19.75 ± 25.54
	8	133.25 ± 26.13	31.75 ± 4.38	101.50 ± 24.12
PBP2				
	Procs	Total Lost	Lost By 1 Receiver	Lost at Sender
Jacobi	4	16.00 ± 2.48	1.33 ± 3.79	14.00 ± 2.48
	8	51.67 ± 27.70	8.67 ± 8.72	42.33 ± 28.80
CG	4	27.33 ± 18.65	1.33 ± 1.43	22.00 ± 17.39
	8	127.00 ± 26.29	18.67 ± 13.68	107.00 ± 31.03

Figure 6.10: Message loss by type (95% confidence intervals shown)

Message loss on a single Ethernet segment can come from three sources. The first is buffer overflow on the sending processor's network interface or too much network contention. There is no mechanism to ensure adequate buffer space for the outgoing socket using UDP. When a message is sent by a user process and the buffer is full, the message is not physically transmitted. With high contention, the exponential back-off algorithm used by the Ethernet controller may exceed its limits. In this case, the message is simply discarded. Secondly, a receiving process' input buffer may be full. This will cause the message to be dropped at the receiver. Other processes may still receive the broadcast message as their buffers may be in different states. Finally, packets may be corrupted in transit, causing a checksum failure. This forces UDP to discard a received packet with no action taken. On a stable network, however, such corruption is very rare. In our experiments we see message loss in two forms.

all or one. When the same message is reported as lost by all non-sending processes, this indicates that either the message was dropped at the sender or the packet was corrupted in transit. When a message is reported lost by only one process, this means the reporting process's receive buffer was full and the message was dropped. In addition, PBP2 showed, in a few isolated cases, messages that were reported missing by all but 2 receiving processes. Figure 6.10 shows the breakdown of each type of message loss. We see that roughly three times as many messages are lost due to sender buffer overflow or corruption.

Procs	Lost	Time (sec.)
4	12	752.88
	16	762.36
	25	762.60
	49	779.31
8	115	658.87
	126	640.09
	139	667.09
	153	657.50

Figure 6.11: Sample execution times for cg

Since messages are inevitably going to be lost, we are interested in the cost associated with detecting and re-sending lost messages. We can informally discuss the effects of loss on the cg program by comparing the number of lost messages to the execution time. Figure 6.11 shows lost message counts and execution times from several executions of cg with four and eight processes. While it is possible to see a correlation between the number of lost messages and execution time for four processes, this correlation is probably coincidental. At eight processes, we see that there are clearly other factors involved. A higher number of messages lost does not automatically lead to worse performance. The type of message lost may be significant. A lost barrier message will likely delay all processes more than a lost

data message. Every process depends on the arrival of the barrier message for continued execution. This is not the case with a single data message. Although, due to FIFO delivery, any messages from the same sender are held up until the lost message is received.

6.5.1 Window Size and Message Loss

jacobi	Procs	Window 16	Window 48	Window 128
	4	16.000000 ± 2.4841	20.666666 ± 16.9092	14.333333 ± 15.1783
	8	51.666668 ± 27.6993	61.333332 ± 20.2321	66.333336 ± 16.1628
CG	Procs	Window 16	Window 48	Window 128
	4	27.333334 ± 18.6448	29.666666 ± 10.0395	28.333334 ± 16.9092
	8	127.000000 ± 26.2896	114.666664 ± 49.3296	111.333336 ± 19.9248

Figure 6.12: PBP2 messages loss versus window size

PBP2 uses variable size windows. The increase in window size allows a process to send more messages at once without requiring any acknowledgments. In some of the benchmarking experiments in chapter 4, increasing the window size increased the number of lost messages due to filling buffers faster and increasing network contention. Figure 6.12 shows PBP2 message loss as a function of window size. While there is some difference among the three window sizes, most noticeably the steady increase in `jacobi` with 8 processes, all of these loss counts fall within each other's confidence intervals. We feel that there is little difference in the loss rate for different window sizes for user applications on PBP. This correlates to the behavior seen using different window sizes in completion time. The differences can be seen on high-demand benchmarks at the PBP level, see chapter 4, but are not visible in user-level applications.

6.6 Conclusions

This chapter presented the experimental results for our BDSM implementation. We discussed the programs in our test suite and the experimental setup and methods used. We used these results to show that the test programs perform as well as similar programs using MPI. These results illustrate the potential usefulness of BDSM as an alternative parallel programming environment for cluster computing on a broadcast capable network. We found that for repetitive programs that had an all-to-all communication pattern BDSM performs well. The cost of collective communication can be reduced by using hardware broadcast operations. We discussed the behavior of these test programs in relation to the underlying network and PBP communication layer.

Comparing the BDSM results to those of the MPI programs we see that collective communication operations that using broadcast can improve performance. In general, computations that are primarily iterative and require shared data among all processes can exploit this improvement. In our test suite this computational model is represented primarily by the cg and `jacobi` equation solvers and the nbody simulation. The `matmult` program shares very little data among all processes since the operands are static. It requires one process to have access to all the results, but there is no interaction among the other compute nodes. Additionally, is a one pass program. There is no repetition. This is shown by BDSM's poor showing next to MPI for this communication pattern.

Chapter 7

Extensions For BDSM

We have presented a new DSM model. BDSM and explored its performance compared to MPI. We also demonstrated its application to several parallel programs. However, there are other applications it can be used for and improvements that can be made. In this chapter we will look at two such extensions. The first involves using BDSM for fault-tolerance. One of our reasons for using fully-replicated weak-memory is fault-tolerance. We explore the potential for using BDSM by deriving a general, fault-tolerant, state-machine service. This state machine provides its service to its client in the presence of failed server nodes. The second part of this chapter addresses the potential scalability issue in two ways. First, we allow memory bound programs to benefit from BDSM by allowing selective segment membership. This allows larger problems to be solved. We then look at methods for allowing more processes by reducing the network traffic. To do this, we ensure updates to a segment are only sent to each process that has joined that segment. By addressing these two issues we show that BDSM can be used for a wider range of applications than were presented earlier.

7.1 Fault-Tolerant Service

Providing a service to client programs is a common use of networking technology. Since the server processor can fail, replicating a server across a number of processors is often desirable. One method of developing such replicated services is to use the state machine approach[56, 78]. The state machine model can be used to implement general, fault-tolerant services. In this section, we present a mapping of one type of state machine service model to the BDSM environment. A simple version of a state machine service is defined by Lamport[56]. The state machine is required to respond to client requests in a causal order. Further, it must ensure that all non-faulty replicas execute requests in the same order despite failures. The original presentation addresses both fail-stop and Byzantine failure modes. Schneider [78] refined and classified this approach. He discusses a number of different techniques that solve the basic problem of ensuring order of requests issued to state machine replicas by clients.

The fault-tolerance requirements of a state machine service can be summarized by two elements. The first is order. Each non-faulty replica processes requests in the same relative order. The second is agreement. Agreement means that all of the functioning replicas see each valid request, with the same time-stamp. Once replicas agree, they can execute the "next" request subject to the ordering requirements. The state machine replicas must all execute the same operations in the same order, thus ensuring the replicated state remains consistent. The order requirement can be summarized by the following two rules:

- **O1:** Requests from a single client are processed in the order issued by the client (program order).

- **O2:** If one client's request causes another client to send a request the first client's request must be handle before the second (causality).

The order requirements, O1 and O2, mean that state machine cannot simply process requests in the order received. Care must be taken to provide causal order among requests by different clients. The agreement requirement is satisfied if the following two conditions hold:

- **A1:** All non-faulty processors agree on the same value for each request (same request in same order).

- **A2:** If the transmitter is non-faulty, then all non-faulty processors use its value.

In this section we develop a state machine model that meets the above criteria using BDSM locations as the communication medium among replicas. We begin with a presentation of the model in general terms. We then present the pseudo-code of each element and discuss the operation of our state machine in detail.

7.1.1 State Machine Model

The service is provided by a system of n replicas, $\mathcal{R} = \{r_0, r_1 ... r_{n-1}\}$, and some clients $\mathcal{C} = \{c_0, c_1, c_2, \dots\}$. The number of clients is unspecified. If the total number of clients is known, then some optimizations may be made. Clients communicate with the server replicas by passing messages over some potentially lossy network. Replicas communicate among themselves using BDSM on an Ethernet LAN. A client c_i issues a request to any single replica in the system. It then awaits at least one response. A replica r_j receives requests and arranges to have all the other non-faulty replicas see the request. The replicas

then agree to act on the requests in the same order. Each replica responds to each request. In an f-resilient system, c_i will receive $n - f$ responses for each request that is processed. Define $nf \subseteq \mathcal{R}$ to be the set of non-faulty processors, such that $|nf| \geq f$ at all times. We are concerned only with fail-stop failures. Since there is no notion of enforced read-only segments in BDSM, there seems to be no, non-cryptographic, way to prevent a malicious program from writing to another replica's shared memory space. Therefore, Byzantine faults are not easily tolerated using BDSM.

In order to allow clients to send requests to any non-faulty replica, and not use a single primary replica to serialize requests, our model is based on real-time clocks. The unique identifier for each request is the time-stamp of the request combined with either the receiving replica's id or the client's id. Clients are assumed to have some unique identifier. We use c_i for this identifier. Since the clients are not required to be numbered, the actual index is irrelevant. In practice, we could use something effectively unique to a machine such as its IP address or MAC address combined with the process identifier of the client process. The system assumes roughly synchronized clocks. For replica r_j, let rc_j be its local clock. Similarly, for client c_i, let cc_i be the client's local clock. Let ϵ be the maximum error between any two process clocks, client or replica. The order that requests are carried out is based on these synchronized clocks. Additionally, we require a reasonable maximum message delay between client and replica, t_c. Since this is potentially large on extremely lossy networks, we can make an arbitrary maximum and refuse to honor requests that fail to arrive in time. A request message that arrives at a replica such that the local time, rc_i, is greater than $ts+\epsilon+t_c$, where ts is the time-stamp of the message, is discarded. The client should timeout and retransmit the request with a new time-stamp. Schneider shows that using real-time

requires the minimum inter-client communication time to be greater than the clock skew, ϵ. This is required to ensure that causally connected requests have noticeably different time-stamps. Since our model requires a client to wait until it gets a reply before proceeding this requirement is easily met. This stop-and-wait operation also allows a client to retransmit a request that either gets lost or has the original replica fail before propagating the request to the group. Among replicas, we define t_r to be the maximum acceptable delay from the time a write is issued until it is applied to the local BDSM copy at some other replica. Any write by r_i will be seen by all $r_j \in nf$ in time less than t_r, if $r_i \in nf$.

For our state machine, we will use the BDSM layer to allow replicas to communicate among themselves. Replicas communicate by writing request messages into BDSM space. Therefore, the buffering system of BDSM is not used. We want writes to be propagated immediately. This also ensures we have PRAM order across all segments. By using BDSM, and therefore PBP, we have reliable FIFO order between any two replicas. PBP does not provide atomic broadcast, so it is possible for a failing node to deliver its request (as an update to a BDSM location) to some set $s \subseteq nf$. This partial broadcast will still take less than t_r time. That is, any replicas that *are* going to receive the update will do so in less than t_r time, just as if the sending replica had not failed.

We assume failed processes are detected. This is not difficult. PBP provides for this. Failed clients have no effect on the system. Clients, since they are not using PBP, must detect replica failure or lost request messages by a timeout. Since clients are effectively stop-and-wait systems, this detection is also not difficult. A client that detects a faulty node, due to a timeout while awaiting a response, simply sends its request to a different replica, with a new time-stamp. Since the client cannot have taken action that causally

precedes another client's request, there is no violation of the ordering requirements when a request is retransmitted. In this case, it is assumed the earlier one simply did not happen.

Messages between client and replica have the following forms. A request message from c_i to any replica is denoted $< REQ, id_c, ts, op, data >$. The fields, other than the type (REQ), are the client's unique identifier (id_c), the time-stamp (ts), the type of operation (op) and any operands required by the operation ($data$). Similarly, reply messages are denoted $< REPL, id_r, id_c, op, data >$. The replica sends back its identifier (id_r), the client's identifier (id_c), the operation performed (op) and any results produced by the operation ($data$). We use structure notation to refer to individual elements of a requests when required. So, for example, given some request r, $r.ts$ is the time-stamp of the request.

7.1.2 Pseudo-code

There are three main components to our state machine model: the client, the BDSM space and the replicas. We start by presenting the client.

7.1.2.1 The Client

Clients make requests to any replica. Faulty replicas are detected by a timeout, at which point another replica may be used. After a request is sent, another cannot be sent until at least one reply to the first request has been received. Since we are assuming fail-stop errors, all non-faulty replicas will be sending the same reply (except for the replica number) so the first to arrive is sufficient. Others can be ignored.

Note that the clock time, cc_i, on a retransmitted message is the current time, not the time of the original request. This ensures that the time-stamp on the request is current

Client$_i$:

send_request: send $< REQ.c_i.cc_i.op.data >$ to $r_j.j \in \mathcal{R}$ then
recv_reply $< REPL.r_h.c_i.op.data >$ from some $r_h.h \in \mathcal{R}$
Timeout: send $< REQ.c_i.cc_i.op.data >$ to $r_k.k \in \mathcal{R}.k \neq j$
recv_reply $< REPL.r_h.c_i.op.data >$ from some $r_h.h \in \mathcal{R}$

Figure 7.1: Client Operation

when received by a non-faulty replica. Once a timeout occurs while sending a request to r_j. a client will not send a request to r_j again. It can simply choose another element of $\mathcal{R}$ confident that at least $n - f$ are non-faulty.

7.1.2.2 The Shared Memory Component

The actual state of the service. the target of client operations such as an NFS file system. is stored locally on each machine. not in BDSM. Initially. this seems counter-intuitive since we have a replicated shared memory space. However. since each process has to decide what action to take independently and then operate on its copy. having the actual state in BDSM is overly redundant. Each process would need its own section of DSM space to represent its copy so the data would be replicated $|\mathcal{R}|$ times at each r_i. Alternatively. a system of mutual exclusion would be needed to ensure one process. only. executed the operation. Failure of the lock holding process. and a subsequent promotion. would then need to be addressed. By keeping the actual state strictly local we avoid this and allow more generality. The service need not act strictly on data that can be stored in memory. but could work on disk files, physical resources and so on. The BDSM space is used to keep a list of pending and resolved requests. This allows replicas to share knowledge about pending requests with one

another in a FIFO, reliable manner.

Shared State:

$\mathcal{Q}[0..n-1][MAXREQ]$ of $(req.id)$
$\mathcal{D}[0..n-1][MAXREQ]$ of $(req.id)$

Figure 7.2: Request lists in shared memory

Shared memory is divided into two lists $\mathcal{Q}$ and $\mathcal{D}$. The lists are further divided so that each replica has it's own section of each list. Replica r_i writes only to its section of $\mathcal{Q}$ and $\mathcal{D}$, the locations $\mathcal{Q}[i][...]$ and $\mathcal{D}[i][...]$. Elements in the list $\mathcal{Q}[i]$ are pending requests as seen by r_i. Those in $\mathcal{D}[i]$ have been performed by r_i. Items in any list are in the form $(req.id)$, where req is a request message and id is the index of the replica which originally received the request. Empty elements may also be used. They are denoted $(null.null)$. When an item no longer appears in $\mathcal{Q}[j].\forall j$, and is in $\mathcal{D}[i]$ it may be removed from $\mathcal{D}[i]$. Removal is done by overwriting a valid entry with the empty entry $(null.null)$. The size of the lists, MAXREQ, is defined for clarity, but need not be strictly defined. If it is, it needs to be greater than the maximum number of clients. If this is unknown the DSM space should be grown dynamically.

7.1.2.3 Request Stability

In order to ensure that each request is applied in the same order at all replicas we need to ensure that

1. there can be no earlier pending requests from the same client and,

2. there can be no earlier pending request from a different client and,

3. the request has been seen by all non-faulty replicas.

The first condition is straight forward due to the stop-and-wait nature of clients. Client c_i can only have one outstanding request at a time. The second condition requires a request to have the earliest time-stamp of any request in $\mathcal{Q}$ and to be time-stamped at least Δ earlier than the current time (at the replica). The delay, Δ, is defined to be $t_r + 2t_r + \epsilon$. The final condition ensures that the request is agreed upon by all non-faulty replicas. It is satisfied when a request is present in $\mathcal{Q}[i] \forall i \in nf$. This property is called *stability*. When a request r is *stable* at r_i it can be executed on the local state and a reply can be generated.

Definition 7.1 *Request r is said to be* stable *at r_i when :*

$$\forall j \in nf : (r.k) \in \mathcal{Q}[j] \wedge (r.k) \notin \mathcal{D}[i] \wedge$$

$$r.ts < s.ts \forall (s, j) \in (\mathcal{Q} \forall j \in nf \cap \mathcal{D}[i]) \wedge$$

$$r.ts + \Delta < rc_i.$$

The comparison $(r.ts < r'.ts)$, for two elements $(r.i)$ and $(r'.j)$, is defined to include the range ϵ for clock skew. The inequality is true if

$$(r.ts + \epsilon < r'.ts) \vee (r'.ts - \epsilon \leq r.ts \leq r'.ts + \epsilon \wedge (i < j \vee (i = j \wedge r.id < r'.id)).$$

Given two request time-stamps, if they are within ϵ then they are ordered arbitrarily by replica number. If they are from the same replica they are ordered by client identifier. This

can be any ordering of the client *id* as long as all replicas can determine the right order. The delay, Δ, ensures that no earlier request is still in transit. The term t_c is used to ensure that all potentially earlier message from clients to replicas have arrived. The term $2t_r$ ensures that all potentially earlier requests that have reached replicas have been seen by all $r_i \in nf$. We need to allow time for the original request to be applied to all non-faulty replicas and for the confirmation copies to be applied. Finally, maximum clock skew, ϵ, is added to cover differences is clocks.

If the number of clients is known and fixed, a request can be declared stable without waiting for the Δ component to become true if there is a later request by each other client in $\mathcal{Q}$. Since there can only be one outstanding request by each client, once each client has a request pending there can be no other requests in transit.

7.1.2.4 The Replicas

Each replica has its own section of each of the lists. Replica r_j reads from $\mathcal{Q}[i].i \neq j$, but does not write to it. This avoids the need for some kind of access control for the shared memory. In a fault-tolerant system where processes fail, we want to avoid having mutual exclusion if possible. Since reads are strictly local operations, there is no waiting for failed processes. A failed process will simply no longer update its sections of $\mathcal{Q}$ and $\mathcal{D}$. Once a process failure has been detected, non-faulty replicas will ignore those areas in further processing.

The basic operation of a replica, r_i, is to wait for requests and to handle them as they become stable. When r_i receives a request, τ, from c_j, it first determines if the request is valid. A valid request must have a time-stamp greater than $rc_i - t_c - \epsilon$ or it has been

Replica$_i$:

[] Recv_request: $r =< REQ.c_j.t.op.data >$ then
$\mathcal{Q}[i][NEXT\_FREE] = (r.i)$:
[] $\exists(r.k)$ in some $\mathcal{Q}[j] \notin \mathcal{Q}[i] \wedge (rc_i < r.ts + \Delta)$ then
$\mathcal{Q}[i][NEXT\_FREE] = (r.k)$:
[] $\exists(r.k) \in \mathcal{Q}[i] \wedge \mathcal{D}[j] \forall j$ then
delete $(r.k)$ from $mathcal Q[i]$:
[] $\exists(r.k) \in \mathcal{D}[i] \wedge \notin \mathcal{Q}[j] \forall j$ then
delete $(r.k)$ from $\mathcal{D}[i]$:
[] if $\exists(r.k)$ that is stable then
execute $r.op$:
send $< REPL.r_i.r.c_j.r.op.data >$:
put $(r.k) in \mathcal{D}[i]$:
[] $\exists(r.k) \in \mathcal{Q}[i] : rc_i < r.ts + \Delta \wedge \neg stable(r.k)$ then
delete $(r.k)$ from $\mathcal{Q}[i]$:

Figure 7.3: Replica operation

delayed too long and should be ignored. Assuming it's a valid request, r_i then writes $(r.i)$ to some free location in $\mathcal{Q}[i]$. This is then seen by all $r_j \in nf$ in time less than t_r. To handle requests, r_i continually scans the $\mathcal{Q}$ locations of other processes for requests it has not seen. If it finds any valid ones, they are copied into $\mathcal{Q}[i]$. When the request with the lowest time-stamp, say (r', j), of any request in $\mathcal{Q}[i]$ is stable, r_i executes $r'.op$, sends a reply to client $r'.id$ and writes (r', j) into a free spot in $\mathcal{D}[i]$. Since all non-faulty replicas see the writing of all requests in $2t_r$ time, all of the non-faulty replicas will find the same lowest request time-stamp and see the same stability conditions. Since the replicas are deterministic, all replicas will execute operations in the same order. The array $\mathcal{D}$ is used to clean up the request list. Once a request has been seen to be moved into $\mathcal{D}$ by all $r_j \in nf$ at r_i, it is removed from $\mathcal{Q}[i]$. Once a request in $\mathcal{D}[i]$ is no longer seen in any $\mathcal{Q}$ it is deleted

from $\mathcal{D}[i]$.

Note that once a request is delivered to a non-fau lty node it will be seen by all others and handled even if the original replica fails having only gotten the write update to one other node. This is true until the message would have become stable. It is possible to conceive of a pathological case where a request is seen by only one non-faulty node which then fails. having passed the request to one other. which then fails. etc. This could allow a request to arrive at all non-faulty nodes having taken $(n - f)t_r$ time. This would mean an old message, which could destabilize a request that has been executed. So once a message passes Δ in age. and has not become stable it is considered invalid.

Once a request gets to a non-faulty node that remains non-faulty it will be seen by all others in the required time. Once a request is seen by all non-faulty replicas. it will eventually become stable and be acted upon. This is true because the conditions for stability will be met. Once r is seen by all replicas. either there is at least one request earlier or there is not. If not. then r is stable as soon as Δ time has passed. If there is an earlier request, then it either becomes stable and gets removed. allowing r to become the earliest. or it fails to become stable (by not being seen in all $\mathcal{Q}$ before Δ time has passed) and is removed. This also allows r become stable.

The replicas operate on requests and clean up used locations by the following rules:

1. A request is not acted upon until it is stable. This means it is the oldest valid request and that it has been seen by all non-faulty replicas. It is copied to $\mathcal{D}[i]$ by each r_i as executed.

2. Once acted upon a request is not removed from $\mathcal{Q}$ until it is seen to have been acted

upon by all non-faulty by appearing in $\mathcal{D}$.

3. Once removed from $\mathcal{Q}$, a request is only totally removed from the system when all non-faulty replicas see that it is removed, by being seeing it in all $\mathcal{D}$ but not in any $\mathcal{Q}$.

7.1.3 Proof

7.1.3.1 Proof of Order and Stability

Schneider showed that the ordering of requests by client time-stamp is effective and satisfies the requirement to have a unique identifier for each request on which to base request order. This is unchanged for us. Roughly synchronized clocks satisfy O1 and O2 if

- No client can issue requests faster than the resolution of the clock can distinguish and
- the clock skew ϵ is smaller than the minimum transmission time between clients.

Our system preserves this order because clients operate in a stop-and-wait fashion. A client cannot have more than one outstanding request. The second condition ensures that causally related events have time-stamps that reflect the causality between them. This is preserved because of the stop-and-wait client semantics as well. A client cannot causally effect another client's request until it has received a reply to its previous request. The time-stamp on the second client's request must then be greater than the first request's time-stamp.

To prove that a stable request is the only one to execute, we define Δ such that once a local clock reaches $ts_i + \Delta$, no earlier request can arrive. We know that no earlier request

can be at another non-faulty node because it would be seen by all non-faulty nodes before $ts_i + t_c + t_r + \epsilon$. and be stable itself in another t_r time units.

If the request reached a replica that failed it is possible for it to take more time than Δ to reach all non-faulty replicas. in which case it will be ignored as being old.

7.1.3.2 Proof of Agreement

We need to show that our model satisfies the agreement properties A1 and A2.

We start with the following lemmas:

Lemma 7.1 *All writes to BDSM locations by any $r_i \in nf$ are seen by all other $r_j \in nf$ in the order issued.*

This follows from the definition BDSM.

Lemma 7.2 *If $r_i \in nf$ then any write issued at time $t = rc_i$ is seen by all $r_j \in nf$ such that $rc_j < t + t_r + \epsilon$.*

This follows from definition of t_r. the use of BDSM (and hence PBP) and roughly synchronized clocks (with the difference between any two clocks bounded by ϵ).

Lemma 7.3 *If r_i becomes faulty while it is writing to BDSM. either some $r_j \in nf$ see the write in less than t_r or none do.*

Since we are assuming only fail-stop failures. in order for this to occur it must happen while the writes are being sent. Either they are received by any other processes before the processor crashes or not. PBP does not guarantee atomic broadcast so the set of receivers is a subset of nf.

Lemma 7.4 *Any valid request r seen by some $r_i \in nf$ is written to $\mathcal{Q}[i]$ and is then seen by all other $r_j \in nf$ at most t_r time later.*

Proof follows from the operation of the replicas. When a request is received by $r_i \in nf$ or seen by $r_i \in nf$ in some $\mathcal{Q}[j]$ and the time is still valid it is written to $\mathcal{Q}[i]$.

Lemma 7.5 *Any unstable request r with $r.ts < rc_i + \Delta$ is considered invalid by $r_i \in nf$.*

Requests that could violate the stability of another request are ignored. Those that prevent the stability of another are removed allowing the later request to become stable. This is designed into the replicas.

Theorem 7.1 *A1 holds for our state machine on BDSM*

Proof: Assume a valid request r arrives at r_i. There are two cases. either $r_i \in nf$ and remains so. or $r_i \in nf$ and fails shortly after receipt. We don't consider $r_i \notin nf$ as the request will never enter the system and the client must timeout and resend it.

1. If r_i remains non-faulty, then A1 holds because of lemmas 1.2. and 4. All $r_j \in nf$ will see (r, i) as written by r_i in at most $t_r + \epsilon$ time. So all non-faulty processes agree on the same value for r.

2. If r_i becomes faulty while transmitting. then. from lemma 3. we have two cases: no process sees the update or at least one $r_j \in nf$ sees the update.

   (a) No other non-faulty process sees r. In this case r is a null operation. The client will need to resend the request.

(b) At least one $r_j \in nf$ sees r in time t_r. If r is still valid at r_j it will be written to $Q[j]$ and seen by all $r_i \in nf$ at most t_r time later. At which point, if still valid, it will be copied into each $Q[i]$ and then become stable. If this extra communication round causes the request to become stale it will be removed. If not it will become stable and be executed.

Theorem 7.2 *A2 holds for state machine on BDSM*

Since we are not concerned with Byzantine failure A2 follows directly from A1. If all non-faulty processes get the value transmitted then they will agree on that value.

We have demonstrated a technique that allows a state machine service to be implemented on the BDSM system. By using such a general technique we illustrate that BDSM can be used for a wide range of highly available service applications, such as a replicated web-server. We feel this serves to show that BDSM is a usable system with potential real-world applications.

7.2 Extending Memory

As designed and implemented our system is fully-replicated. The entire address space is resident at every processor. While this is useful for fault-tolerance and has been effective for the compute bound test programs in chapter 6, it is not always desirable. Some parallel programs require more memory than is available on a given workstation. As a potential extension of BDSM, we would like to address this issue. There are two issues involved: extending memory usage and increasing scalability. The first issue can be addressed by allowing only those processes that need a segment to join it. Processes that don't join a

segment would not allocate local memory space for it. The second issue is to reduce the communication overhead. Once we allow selective segment membership, we would like to avoid the overhead of reliable message passing for updates to processes that have not joined a given segment. While all messages are broadcast, requiring reliable, FIFO delivery of messages, and therefore acknowledgments of some kind, is wasteful for processes that will ignore the message.

7.2.1 Expanding Memory Usage with Selective Join

By allowing segments to be created yet not joined by all processes, memory can be partitioned. Only processes that actually join a given segment would allocate space for that segment. In this way, only processes that need access to those memory locations would use real memory storing them. Other subsets of the processes could join other segments and thereby extend the amount of memory seen by the whole program. This first step addresses the issue of memory bound computations. It permits only those processes that need a segment to join it. Processes that don't join a segment would not allocate local memory space to it. As currently implemented, when a process receives a create segment message it allocates the space then so that it can begin processing any updates that arrive after creation but before the local process joins the segment. Reversing the semantics would mean a barrier or some form of consistency check would be needed when a process joins a segment. This synchronization would be needed to ensure that no writes can be made until every process joins a given segment. A barrier placed after segment creation and join suffices to address this issue.

This modification is simple and requires little change to the existing implementation.

It allows a computation to have, not only, the illusion of shared memory, but also, that of more physical memory than is present on any one processor. Normally, when fully-replicated, BDSM provides physical memory equal to the minimum physical memory of any processor used. Issues of different memory capabilities on a heterogeneous network can be addressed. Processors with more physical memory can be assigned processes that need access to more segments. Those processors with more limited physical memory can be assigned processes that join fewer segments.

7.2.2 Improving Scalability

Extending memory by allowing selective join increases the scalability of the BDSM system. It allows for larger programs that would not fit in the physical memory of any one processor. However, since we use hardware broadcast for each update, we still have updates being sent to all processes. A process that receives an update for a segment it has not joined simply ignores it. The problem is that these ignored messages are sent by PBP, so are sent reliably to processes that don't need them. Additionally, the message has to be delivered to the BDSM implementation before being ignored. When used on a system with a large number of processes, the cost of broadcast reliability for messages that only need to reach a subset will become higher. A broadcast on an Ethernet segment is effectively the same as a point-to-point message. However, requiring acknowledgements for messages that are to be ignored does use more network bandwidth. We would like to spend resources ensuring the delivery and order of updates only to those processes that need the update. Since we are using broadcast, each message sent is still seen by all processes. Messages that aren't important to a given process could be ignored at a lower level and will not need to be acknowledged.

To make the system perform in this fashion we will use an instance of PBP for each segment. The creation and joining of a segment will contain the group membership protocol of PBP initialization. Since PBP is a stand-alone system we simply need to start PBP with each process that joins a segment. Each instance of PBP will use a different port number so the messages that a process can ignore will be dropped at the kernel level. Each process will then perform inputs on any PBP queue that has messages available. Some form of PBP delivery multiplexing can be used to determine which incoming queues have messages.

An alternative implementation method would be to redesign PBP to use IP Multicast. Each segment would have a multicast group associated with it. The PBP system would then provide FIFO service among members of each group. In this case, there would effectively be an instance of the PBP protocol for each multicast group. With hardware that effectively filtered IP multicast packets, non-members would be only minimally effected by messages exchanged among members of a given group.

This system needs to preserve the BDSM requirements from chapter 2. These requirements are

1. Writes by a process to a given segment appear in program order.

2. Synchronization operations issued by any process appear in program order.

3. Synchronization operations appear in program order with respect to all writes issued by a process. Writes before the synchronization appear before and those after, after.

To satisfy the first requirement we rely on PBP. Each segment will have an instance of PBP for group communication among all joined processes. Since updates to each segment

use this FIFO channel provided by PBP. the requirement that writes to a given segment be in program order is preserved. This does not require any changes.

Ensuring order for synchronization operations requires an additional communication channel. While the basic system of using a PBP instance per segment is straightforward. its effects on the synchronization coherence model are not. In order to ensure that we maintain BDSM coherence. we require synchronization operations to be in program order. To this end we will use one global instance of PBP for all processes. This main PBP instance will be used for synchronization operations. This will satisfy the BDSM condition that synchronization operations be in program order with respect to other synchronization operations. Additionally. since all synchronization in BDSM is global. either barriers or broadcast locks. each process needs to be able to communicate with each other regardless of segment membership. A single channel for all processes allows this global message passing to take place.

The last requirement for the BDSM model is that updates by a given process to all segments it has joined be seen in FIFO order relative to each synchronization operation sent by that process. When the BDSM layer of a process performs a barrier. it will send a message with the barrier number down each of its PBP connections. Then. it will send the barrier message on the global PBP channel. This effectively makes a checkpoint on each of its PBP channels at the point in the program order that the barrier was called. Other processes will only consume messages from a given process up to an not beyond a barrier marker in a segment PBP queue until the consuming process has reached the barrier marker for that process on each input queue and crossed the barrier. In this way. a process will not see any writes made by another process after a barrier before crossing it and vice-versa. We

call this the *barrier marker* system. These messages are similar to the 2-way flush messages in flush channel communication[7. 24]. We have defined this model to work for barriers because they have clear simple semantics. Using this for the distributed locks in BDSM would require a similar protocol for each lock message. those responding to lock requests as well acquire requests.

7.2.3 Barrier Marker System

A barrier in BDSM. as seen by one process. consists of receiving $n-1$ barrier messages and issuing a barrier message itself. These barrier messages consist of $(bar.b.i)$. where b is the barrier number and i is the process sending the barrier message. The barrier number is used to differentiate among barriers. For this extension. the protocol is similar. but more involved. Now. a process sends a message. in this case called a barrier marker and denoted $(brm.\ b.\ i)$. down the channel for each segment it has joined. Then. it sends the regular barrier message down the global channel. To cross the barrier it must still receive a barrier message from each other process. It must also receive a barrier marker from each segment for each other process that has joined that segment. Definition 7.2 shows the barrier condition. BC. which must be satisfied for a process to cross a barrier. Additionally. once a barrier marker from some process has been received for a given segment. no other messages from that process for that segment may be consumed until the receiving process satisfies BC. They must be queued locally and handled after the the barrier is crossed. This condition is called WC. see definition 7.3.

Joining a segment, s_i, provides a communication channel. c_i. that delivers messages to all processes that have joined s_i in the order sent by each sending process. This is provided

by the use of PBP as the communication layer for each segment. There is no order guarantee between messages sent on different channels.

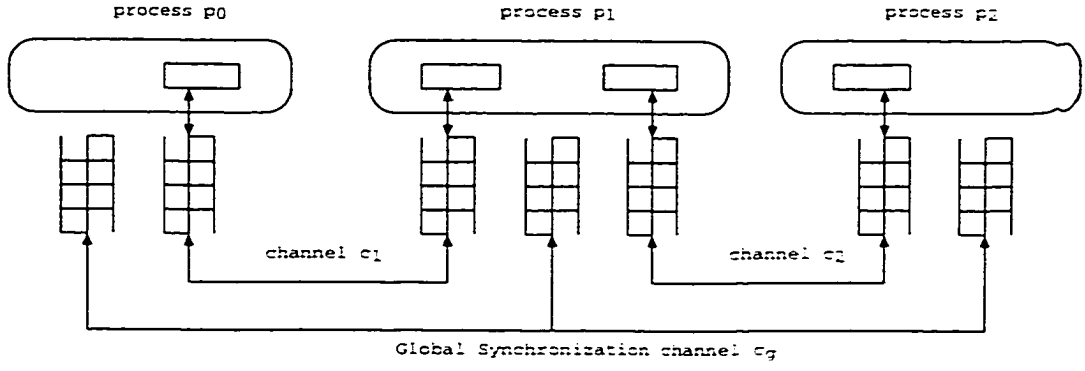


Figure 7.4: BDSM using multiple PBP channels.

Consider three processes p_0, p_1 and p_2, shown in figure 7.4. Each is a member of the global PBP communication channel, c_g. Additionally, two processes have each joined segments s_1 and s_2, which have PBP channels c_1 and c_2, respectively. Process p_1 has joined both memory segments. When p_1 reaches a barrier, say b_0, it sends a barrier marker message, (bm, b_0, i), down each memory segment channel and then a normal barrier message, $(bar, b_0.i)$, down the global channel. Process p_1 will not cross the barrier until it has received a message $(bar, b_0.0)$ and $(bar, b_0.2)$ from c_g. Additionally, it must receive $(bm, b_0.0)$ from c_1 and $(bm, b_0.2)$ from c_2. Once $(bm.b_0.0)$ is received by p_1 from c_1, no other message from p_0 may be handled from c_1 until the barrier is crossed. Similarly, once p_1 has received $(bm, b_0.0)$ from p_2 on c_2 it will not handle any other messages from p_2 on c_2. When p_0 reaches the barrier, it needs to wait for a barrier message from each other process on c_g and a marker on c_1 from p_1. It doesn't need anything other than the barrier message on c_{cg} from p_2 because p_0 and p_2 do not share segment membership.

Definition 7.2 *Barrier Condition (BC): A process p_i may complete a barrier operation*

when it has

1. *reached the barrier itself.*
2. *received a matching barrier message from each process in the system on c_g. and*
3. *received a matching barrier marker for each segment s from each other member of s for each s_i of which p_i is a member.*

Definition 7.3 *Wait condition (WC): A process p_i may not receive a regular message for any segment s from some p_j if p_i has received a barrier marker from p_j for s and has not satisfied BC for that barrier.*

7.2.4 Proof

In this section, we prove the barrier marker system is effective. This requires proving liveness and safety of the barriers themselves and that the ordering requirements of BDSM are preserved. Liveness and safety were shown for barriers in theorem 5.1. This form of barrier, with barrier markers, behaves the same way with respect to safety and liveness. A process cannot cross a barrier until it satisfies BC, which includes receiving an appropriate barrier message from each other process. Liveness is also assured because each process must send its barrier marker messages and P1 ensures they will all be delivered.

The first of the BDSM requirements, from chapter 2, is that all writes by one process to a given segment be see by all others in program order. The barrier marker protocol has no effect on the order of writes. Therefore, this requirement is met.

The second requirement is that synchronization operations occur in the program order of each process. Since there is a single PBP channel shared by all processes and this channel is used for all barrier messages. these messages will be in program order.

To prove the third BDSM order requirement we will break it into two parts.

1. No write update messages sent by p_i after a barrier can be seen by any p_j before p_j crosses the barrier.

   Assume the opposite. Some message m. sent after a barrier b in p_i is seen by some p_j before p_j crosses b. In order for it to have been seen it must have been delivered. And in order to be delivered it must have been received on some c_k. Since p_i has reached b it has sent a barrier message on c_g and a barrier marker message on each c_k it has joined. Also, since m has been sent after all the messages for b. the barrier marker message must have been sent on c_k before m. From P1. it must also be delivered before m. Since p_j has not satisfied BC. or it would also have crossed the barrier. it cannot have seen any messages from p_i on c_k after seeing the barrier marker. from WC. Therefore it cannot have seen m.

2. All messages sent by p_i before the barrier will be seen by each p_j before p_j crosses the barrier.

   Assume the opposite. Some message m sent by p_i before b is not seen by some p_j after it crosses the barrier. Since m must have been sent on c_k before the barrier marker for b by p_i, it must have been delivered after it in order to have not been seen by p_j when it satisfied BC. This is a clear violation of P1. Therefore m must have been seen before p_j satisfied BC and crossed the barrier.

The barrier marker system will preserve the ordering requirements of BDSM. It allows a program to be more selective in its us of memory thus allowing more flexibility. This extension addresses issues of scalability by making BDSM more efficient for larger numbers of processes by reducing network utilization.

7.3 Conclusions

In this chapter we presented two different ways that the BDSM system can be extended to make it more useful to many applications. The first is to use BDSM for fault-tolerant services. The general state machine model allows any client/server applications to be designed for BDSM. The second is deals with scalability. By addressing some of the scalability issues of BDSM we show that is can be used for larger problems. Many applications are implemented in parallel to acquire more physical memory than is on a single processor. The extension presented here allow this to be done by programs using BDSM as well. The barrier marker system allows programs to make use of more processes by making the overall system more efficient. These two results serve to illustrate the potential utility of a DSM system designed for the use of broadcast on a common clustered computational platform.

Chapter 8

Conclusions

Distributed shared memory provides the illusion of a shared address space to processes on systems with no shared memory. Software DSM systems provide this service to processes on separate workstations connected by a network. For efficiency, many of these systems provide weak memory semantics. While it has been argued that these weak memories should not be used on hardware multiprocessor systems[46], the performance gains are often still necessary on a cluster computing system. There is a significant amount of work being done to improve software DSM systems[49, 55, 26, 76, 28, 17, 75, 10, 54]. The need for simpler implementations that still provide good performance has been expressed[88]. We have developed a system that relies on a simple protocol to provide weak DSM to processes sharing an Ethernet segment.

In this chapter we present some of the conclusions we have drawn from this work. We start by looking at a few potential directions for future work. This addresses some of the issues and weaknesses of the system. We then summarize the results we presented. And discuss how we have attained the goals we laid out in the introduction.

8.1 Future directions

We are pleased with the results our BDSM system has shown to date. However, there is room for improvement. Some issues to address in the future are write detection, using TCP, and implementation of the extensions in chapter 7. There are also several refinements of PBP we would like to explore. We discuss each in this section.

Write detection is of major concern because without it the system cannot be made truly transparent[96]. Currently, writes must be a function call that lets the BDSM thread perform the update. Using some intermediate layer that catches writes without this overhead and propagates an update would be helpful. Systems like SHRIMP [50] use modified hardware that automatically sends writes to the network, similar to a write-through cache. It may also be possible to use the memory management structures to protect memory location on a fine granularity. The Region-Trap library [21] would facilitate this. A handler function would still be called for each write so there may be little performance gain. However, it would complete the illusion of a shared address space by providing true transparency at the user-level.

The BDSM system is not required to use broadcast. It would be possible to implement a set of TCP connections among processes and make the operations send to each connection. The system would still need to send each message to each other process to preserve the semantics of BDSM, but this would allow the system to work on wider networks. It would also allow us to make a real measure of the benefits of using hardware broadcast, by comparing the two versions. Implementing PBP using IP Multicast would have a similar effect, and still allow the use of hardware broadcast where available.

To improve scalability we would like to implement the extension to allow multiple PBP instances. This will give the BDSM system wider utility by allowing larger problems. Additionally, since we are also interested in fault-tolerance, a prototype state machine implementation would server to better illustrate the potential for highly-available computing using BDSM.

In the future we would like to explore some issues regarding PBP as well. It Would be interesting to allow PBP to have dynamic timeouts. The notion of a round-trip time (RTT), used by protocols like TCP to change various timeout values, is less well-defined for a broadcast system. We can imagine a form of RTT that is similar to that used for point-to-point protocols. The time would be based on when an outgoing message buffer was reclaimed after being allocated. With such a system, we may be able to improve the performance of PBP further by more accurately timing events.

Another possible improvement to PBP would be to allow dynamic connections. Currently, the system is limited to those processes that participate in the group registration process at startup. Processes can be removed due to failure or voluntary exiting. This change in membership is currently one way only. Processes can be removed but not added. Allowing the group membership to grow would allow failed processes to be replaced without restarting the entire group. This would make PBP more applicable to fault-tolerant computing by allowing process recovery.

8.2 Conclusions

We have developed a weak form of DSM tailored to be efficient in a common networking situation. Many clustered or networked computing environments use some form of Ethernet as a communication medium. Our system uses the inherent broadcast ability of this hardware to perform efficient all-to-all communications. Writes in BDSM are distributed as broadcast updates. We allow a weak enough model that there is no need to have global, or even causally, ordered message-passing. This means no need for extra messages or the serialization of broadcasts.

We have overcome some of the problems of using non-causal memories. Many such systems are too weak to be programmed effectively. PRAM and Slow memory are examples of such weak, non-causal memories. Synchronization operations based on memory locations do not have enough power. We solve this by using synchronization at a lower, message-passing level. Our synchronization operations are broadcast by the communication layer rather than being DSM level writes to memory locations.

We have developed a test suite of common parallel computations. These programs are used as comparisons to MPI, a common message-passing alternative. We show that BDSM can be a viable alternative to message-passing on a LAN because our test program performance is comparable to that of their MPI counterparts. We found that for true collective communication operations, such as are required for iterative methods, the use of broadcast scaled better than MPICH. The MPI implementation uses TCP connections on a network of workstations. Our system shows better results because the all-to-all communication is cheaper with broadcast operations. This leads us to the conclusion that for programs with

significant numbers of collective communication operations. a broadcast DSM system is a viable alternative to message-passing.

In the course of our work. we have developed a FIFO. reliable broadcast system and implemented it as a library. This PBP system provides efficient use of the normally lossy UDP broadcast on a single Ethernet segment. Additionally. we have tested the performance of PBP versus TCP for throughput. These results show the expected increase in effective throughput for more then 2 processes. We have also compared these throughput results to published results of a different reliability protocol that can take advantage of hardware Ethernet broadcast. RMP. PBP compares favorably at the expense of total order. However. even for a one sender situation. where sender order is total order. PBP performance is closer to the hardware limits than RMP.

In chapter 7. we presented two extensions to the BDSM system. The first is an application of BDSM to a fault-tolerant server model. We show that BDSM can be a general service provider that provides high availability in the presence of message loss and failed processes. A second extension was presented that addressed some issues of scalability in BDSM. We show how BDSM can allow for larger. memory bound computations by not fully-replicating memory. Further. we have shown a scheme to reduce the PBP communication traffic to only those processes that need each update. These two proposals show that a broadcast DSM system can be applied to a larger range of applications than we have actually implemented.

We have developed a weak DSM model that does not require global. or causal ordering of the updates. This system can be used effectively. due to strong synchronization operations. Using Ethernet broadcast capabilities can reduce the cost of all-to-all communication. We

have demonstrated how this can be done in a reliable fashion to implement a weak update based DSM system. Through experimentation we have shown that broadcasting updates can be a competitive method of interprocess communication on a LAN for programs with appropriate communication patterns.

Appendix A

Sample Test Code

In this appendix we present shortened versions of the jacobi code from the test suite. We show both the BDSM version and the MPI version for comparison.

A.1 BDSM Jacobi Code

```
/* FILE: jacobi.c
 * Written by :
 * Philip R. Auld
 * Dept. of Computer Science
 * College of William and Mary
 *
 * Jacobi linear equation solver for dsm
 * Solves for x in Ax = b
 *
 * Creation Date: 7/3/98
 * Last Modification Date: 10/27/98
 *
 * Changed to use shared mem for all data.
 *
 *
 ************************************************************************/
#include    <stdio.h>
#include    <stdlib.h>
```

```
#include     <unistd.h>
#include     <strings.h>
#include     <math.h>
#include    <sys/time.h>

#include     "dsm.h"

/* dsm info */
int d_id_data;
int d_id_res;
int proc_num;
/* these will be starting location in dsm segment of each matrix */
int vector_x, vector_d;
/* Pointers for direct DSM access */
float * x_ptr;
float * d_ptr;

int debug = 0;
int doprint =0 ;
int test_result = 0;
int use_file = 0;
int rand_seed = 0;
int max_iter = 1000;
int matsize = 4 ;
int numproc = 1;
int done_loc;

/* target conversion bound */
float epsilon = 0.001;

/* these will hold problem constants A and b */
int mat_a;
int *a_ptr;
int vec_b;
int *b_ptr;

/* count interations */
int num_iterations = 0;

/* which barrier, will alternate between 1 and 0. */
int current_barrier ;

/* test the done value. Returns < 1. (hopefully 0) if false
 *   1 or > if done is true
 *
```

```
 ********************************************************/
float
read_done()
{
   float val;
   dsm_read(d_id_res, &val, done_loc);
   return val;
}

void
show_solution()
{
  /* print solution, including current x values */
}

void test_solution()
{
 /* if we are right then Ax should be pretty close to the original b*/
}

void
show_problem()
{
/* prints problem values and "x[n]" for each x value */
}

/* we use a strongly three or five diagonal matrix to help ensure convergence */
int
get_diag_data(int * a, int *b, int size)
{
  /* generate  ....0,-1,-1.4,-1,-1.0.... type matrix  */
}

/* initialize all constant data, Generate diagonal matrix and random b
  Or read input from file if given */
void init_matrix()
{
  int i;
  int num_to_send;
  int * matrix_a,* vector_b;
  FILE * input_data;

  matrix_a = (int *) malloc ( (matsize*matsize+ matsize)*sizeof(int));
  vector_b = &matrix_a[matsize*matsize];
  get_diag_data(matrix_a, vector_b, 5);
```

```
  num_to_send = (matsize * matsize) + matsize;
  /* for large number of adjacent writes this is much more efficient */
  dsm_bulk_write (d_id_data_mat_a, &matrix_a[0] , num_to_send, DSM_WRT_REL);
  free (matrix_a);
}

/* Setup actual pointer into the dsm segemt for reading directly */
void
get_pointers()
{
  if ((x_ptr = (float*) dsm_ptr_read(d_id_res, vector_x)) == NULL){
    /*ERROR */
    dsm_exit();
    return;
    }
  if ((d_ptr = (float *)dsm_ptr_read(d_id_res, vector_d)) == NULL){
    /*ERROR */
    dsm_exit();
    return;
  }
  if ((a_ptr = (int *)dsm_ptr_read(d_id_data, mat_a)) == NULL){
    /*ERROR */
    dsm_exit();
    return;
  }
  if ((b_ptr = (int*)dsm_ptr_read(d_id_data, vec_b)) == NULL){
    /*ERROR */
    dsm_exit();
    return;
  }
}

/* given i this returns the new value for x[i] based on the current
 * values and matrix A and vector b
 ************************************************************/
float calculate_value(int i)
{
  int j;
  int k;

  float partial_solution;
  float lower_sum = 0.0;
  float upper_sum= 0.0;
  int row = i * matsize;
```

```
  /* we read the values directly from the dsm segment using pointers */
  partial_solution = b_ptr[i];
  for ( j = 0 ; j < i; j ++){
    lower_sum += x_ptr[j] * a_ptr[row +j];
  }
  for ( k = i+1 ; k < matsize; k ++){
    upper_sum += x_ptr[k]* a_ptr[row+k];
  }
  partial_solution -= lower_sum;
  partial_solution -= upper_sum;
  partial_solution / (float) a_ptr[row + i];

  return partial_solution;
}

/* test for convergence. Finds max d[i] and compares it to
 * the desired epsilon convergence bound.
 ************************************************************/
int converged()
{
  int x;
  float curr_max =0.0;

  for( x = 0 ; x < numproc; x ++){
    curr_max = max(d_ptr[x], curr_max);
  }
  if ( curr_max < epsilon)
    return 1;
  return 0;
}

/* runs the jacobi algorithm */
int
solve_problem ( )
{
  int       x,k;
  int retval = 1;
  int global_start;
  int num_to_compute;
  float * temp_values;
  float current_max;
  int count;
  int curr_loc;
  float temp_done;
```

```
num_to_compute = matsize / numproc:
k = matsize % numproc:
if ( k > proc_num)
  num_to_compute ++:

temp_values = (float * ) malloc ( sizeof(float)* num_to_compute):
if (temp_values == NULL){
  dsm_exit():
  exit (-1);
}

/************************** BARRIER ************************/
dsm_barrier(current_barrier. &numproc):
current_barrier = !current_barrier:

global_start = proc_num * num_to_compute:

while (read_done() < 1){

  /* calculate phase. no dsm writes */
  current_max = 0.0:

  for (count = 0 : count < num_to_compute: count ++ ){
    curr_loc = global_start + count:
    temp_values[count] = calculate_value(curr_loc):
    current_max = max(current_max.
                ((float) fabs((float)(temp_values[count]- x_ptr[curr_loc])))):
  }

  /************************** BARRIER ************************/
  dsm_barrier(current_barrier, &numproc):
  current_barrier = !current_barrier:

  /* write back phase, reads are unsafe here */
  dsm_write (d_id_res. vector_d + proc_num. &current_max. DSM_WRT_REL):
  for (x = 0; x < num_to_compute: x++){
    curr_loc = global_start + x;
    dsm_write (d_id_res, curr_loc, &temp_values[x]. DSM_WRT_REL):
  }

  /***************************** BARRIER ***********************/
  dsm_barrier(current_barrier, &numproc):
  current_barrier = !current_barrier:

  if ( proc_num == 0){
```

```
    if (num_iterations ≥ max_iter){
      temp_done = 1;
      retval = 0;
    }
    else
      temp_done = (float)converged();
    dsm_write (d_id_res, done_loc, &temp_done, DSM_WRT_REL);
  }

  /*********************************BARRIER*****************************/
  dsm_barrier(current_barrier, &numproc);
  current_barrier = !current_barrier;

  num_iterations ++;
 }/* while not done */

 if(proc_num == 0)
  printf("%d ", num_iterations);
 return retval;
}


/* main function sets up dsm, attaches to a dsm segment and then calls other
 *  functions to actually solve the problem.
 ************************************************************************/

int
main(int argc, char *argv[])
{
 int s_flag;
 int numlocs_x, numlocs_data;
 int c;
 struct timeval start_time, end_time;
 long temp_time_sec, temp_time_usec;
 extern char * optarg;

 if ( gettimeofday(&start_time , NULL) <0){
  perror ("gettimeofday");
  exit(1);
 }

 /* start program different machines*/
 if ((s_flag = dsm_startup(&argc,argv)) <0){
  dsm_perror("startup");
  exit (-1);
```

```
}
strncpy(data_file, DATAFILE, 128);

while ((c = getopt(argc, argv, "s:p:i:m:f:e:olthd")) ≠ -1) {
 /* Process arguments */
}
if (rand_seed == 0)
  rand_seed = getpid();

/*Setup number of location we will need. Point some integers into the DSM space to mark
  location of different arrays */
numlocs_x = matsize + numproc + 1;
vector_d = matsize;
vector_x = 0;
done_loc = numlocs_x -1;
numlocs_data = matsize*matsize + matsize;
mat_a = 0;
vec_b = matsize*matsize;
current_barrier = 0;

/* actually start DSM */
if ((proc_num  = dsm_init(&numproc_s_flag)) < 0 ){
  dsm_perror("BADNESS\n");
  exit(-1);}

/* create 2 DSM segments . one of ints and one of floats */
if (proc_num == 0){
  d_id_res = dsm_seg_at (numlocs_x.sizeof(float),1234,DSM_CREATE);
  if (d_id_res <0){
    dsm_perror("seg_at!");
    exit (-1);
  }
  dsm_sleep(5);
  d_id_data = dsm_seg_at (numlocs_data.sizeof(int),4321,DSM_CREATE);
  if (d_id_data <0){
    dsm_perror("seg_at!");
    exit (-1);
  }
}
else {
  /* attach to the 2 segments greated by process 0 */
  while((d_id_res = dsm_seg_at (numlocs_x.sizeof(float),1234,DSM_JOIN)) <0)
    dsm_sleep(2);
  if (d_id_res <0){
    dsm_perror("seg_at!");
```

```
    exit (-1);
  }
  while((d_id_data = dsm_seg_at (numlocs_data.sizeof(int).4321.DSM_JOIN)) <0)
    dsm_sleep(2);
  if (d_id_data <0){
    dsm_perror("seg_at!");
    exit (-1);
  }
}
get_pointers();
if(debug){
  printf("got pointers,calling barrier %d \n", current_barrier);
  fflush(stdout);
}
/* makse sure every one has started before we start writting initialization data */
dsm_barrier(current_barrier, &numproc);
current_barrier = !current_barrier;

if (proc_num == 0)
  {
    fprintf (stderr,"done with sys_init\n");
    if ( gettimeofday(&end_time , NULL) <0){
  perror ("gettimeofday");
  exit(1);
    }
    temp_time_sec = end_time.tv_sec - start_time.tv_sec;
    temp_time_usec = end_time.tv_usec  - start_time.tv_usec;
    printf ("%ld ", temp_time_sec*1000000+ temp_time_usec);
    init_matrix();
    fprintf (stderr,"done with init\n");
    if ( gettimeofday(&start_time , NULL) <0){
      perror ("gettimeofday");
      exit(1);
    }
    temp_time_sec = start_time.tv_sec - end_time.tv_sec;
    temp_time_usec = start_time.tv_usec  - end_time.tv_usec;
    printf ("%ld ", temp_time_sec*1000000+ temp_time_usec);
  }

if (solve_problem() == 1)
  fprintf(stderr,"Solved\n");
else
  fprintf (stderr,"max iterations reached\n");

if (proc_num == 0)  {
```

```
    if ( gettimeofday(&end_time . NULL) <0){
      perror ("gettimeofday"):
      exit(1):
    }
    temp_time_sec = end_time.tv_sec − start_time.tv_sec :
    temp_time_usec = end_time.tv_usec − start_time.tv_usec :
    printf ("%ld\n ", temp_time_sec*1000000+ temp_time_usec):

  }

  dsm_remove(d_id_data):
  dsm_remove(d_id_res):

  dsm_exit():
  dsm_bcast_stats(proc_num. stdout ):

  return 0:
}
```

A.2 MPI Jacobi Code

```
/* File: jac_mpi.c
*
* Original version from "Parallel Programming with MPI".
* by P. Pacheco, Morgan Kaufmann Publishers,
* Los Altos. CA 94022. USA, 1997.
*
* Modifed by Philip R. Auld
*************************************************************************/

#include     <stdio.h>
#include     <stdlib.h>
#include     <unistd.h>
#include    <sys/time.h>
#include     <strings.h>
#include     <math.h>

#include     "mpi.h"

#define SWAP(x,y) {float* temp; temp = x; x = y; y = temp;}
#define MAX_DIM 1024

int rand_seed = 1;
int max_iter = 1000;
int matsize = 4;
/* target conversion bound */
float  epsilon = 0.001;

typedef int MATRIX_T[MAX_DIM][MAX_DIM];

int
parallel_jacobi(MATRIX_T  A_local,float x_local[],float b_local[], int n,
                float tol, int max_iter, int p, int my_rank);

void read_matrix(MATRIX_T A_local, int n,
         int my_rank, int p);

void read_vector(float x_local[], int n, int my_rank,
         int p);

void Print_matrix(char* title, MATRIX_T A_local, int n, int my_rank, int p)
{
/* print A_local to stdout */
```

```
}

void Print_vector(char* title, float x_local[], int n,int my_rank, int p)
{
/* print X to stdout */
}

void get_diag_mat(MATRIX_T a, int n, int size){
/* initialize a with ....0.-1.-1.4.-1.-1.0.... type diagonal matrix */
}

void get_rand_vec(float *b, int size)
{
/* initializa b with random values */
}

void
show_problem(MATRIX_T a, float * b, int n);

void
show_solution(MATRIX_T a, float * x_loc,float * b, int n);

void test_solution(MATRIX_T a, float *b, float *x, int n);

/* data in data segment to avoid stack overflow */
MATRIX_T   A_local;

void
main(int argc, char* argv[]) {
  int       p;
  int       c;
  int       my_rank;
  float     x_local[MAX_DIM];
  float     b_local[MAX_DIM];
  int       converged;
  struct timeval start_time, end_time;
  double temp_time_sec, temp_time_usec;

  if ( gettimeofday(&start_time , NULL) <0){
    perror ("gettimeofday");
    exit(1);
  }

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == 0) {
  while ((c = getopt(argc, argv, "s:m:e:i:othd")) ≠ -1) {
    /* process arguments */
  }
}

MPI_Bcast(&matsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&epsilon, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&max_iter, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (my_rank == 0)
  {
    fprintf (stderr,"done with sys_init\n");
    if ( gettimeofday(&end_time , NULL) <0){
  perror ("gettimeofday");
  exit(1);
    }
    temp_time_sec = (double)(end_time.tv_sec - start_time.tv_sec);
    temp_time_usec = (double)(end_time.tv_usec  - start_time.tv_usec);
    printf ("%10.0f ", temp_time_sec*1000000+ temp_time_usec);
  fflush(stdout);
  }

read_matrix(A_local, matsize, my_rank, p);
read_vector(b_local,matsize, my_rank, p);

if (my_rank == 0){
  fprintf (stderr,"done with init\n");
  if ( gettimeofday(&start_time , NULL) <0){
    perror ("gettimeofday");
    exit(1);
  }
  temp_time_sec = (double)(start_time.tv_sec - end_time.tv_sec);
  temp_time_usec = (double)(start_time.tv_usec  - end_time.tv_usec);
  printf ("%11.0f ", temp_time_sec*1000000+ temp_time_usec);
 fflush(stdout);
}
converged = parallel_jacobi(A_local, x_local, b_local, matsize,
                            epsilon, max_iter, p, my_rank);
if (converged){
  if(doprint)
    Print_vector("The solution is", x_local, matsize, my_rank, p);
  if (test_result)
```

```
    test_solution(A_local, b_local, x_local, matsize);
  }
  else
    if (my_rank == 0)
      fprintf(stderr,"Failed to converge in %d iterations\n", max_iter);
  if(my_rank ==0){
    if ( gettimeofday(&end_time , NULL) <0){
      perror ("gettimeofday");
      exit(1);
    }
    temp_time_sec = (double)(end_time.tv_sec - start_time.tv_sec) ;
    temp_time_usec = (double)(end_time.tv_usec - start_time.tv_usec) ;
    printf ("%12.0f\n", temp_time_sec*1000000+ temp_time_usec);
  }
  MPI_Finalize();
} /* main */

/*****************************************************************/
/* Return 1 if iteration converged. 0 otherwise */
/* MATRIX_T is a 2-dimensional array            */
int
parallel_jacobi(MATRIX_T  A_local,float x_local[],float b_local[], int n,
                float tol, int max_iter, int p, int my_rank)
{
  int     i_local, i_global, j,k;
  int     n_bar;
  int     iter_num;
  float   x_temp1[MAX_DIM];
  float   x_temp2[MAX_DIM];
  float*  x_old;
  float*  x_new;
  float max_diff, diff_local;
  int x, done = 0;
  float upper_sum, lower_sum, partial_solution;

  n_bar = n/p;
  /* Initialize x */
  MPI_Allgather(b_local, n_bar, MPI_FLOAT, x_temp1,
            n_bar, MPI_FLOAT, MPI_COMM_WORLD);
  x_new = x_temp1;
  x_old = x_temp2;
  iter_num = 0;
  do {
    iter_num++;
```

```
    diff_local = 10000.0;
    /* Interchange x_old and x_new */
    SWAP(x_old, x_new);
    for (i_local = 0; i_local < n_bar; i_local++){
      i_global = i_local + my_rank*n_bar;
      upper_sum = lower_sum = 0.0;
      partial_solution = b_local[i_local];
      for ( j = 0 ; j < i_global; j ++){
    lower_sum += x_old[j] * A_local[i_local][j];
      }
      for ( k = i_global+1 ; k < n; k ++){
    upper_sum += x_old[k]* A_local[i_local][k];
      }
      partial_solution -= lower_sum;
      partial_solution -= upper_sum;
      partial_solution / (float) A_local[i_local][i_local];
      x_local[i_local] = partial_solution;
      diff_local = max (diff_local, fabs(x_local[i_local] - x_old[i_global]));
    }
    MPI_Allgather(x_local, n_bar, MPI_FLOAT, x_new,
                n_bar, MPI_FLOAT, MPI_COMM_WORLD);
    max_diff = 0.0;
    for ( x = 0 ; x < matsize; x ++){
      max_diff = max ( max_diff, fabs(x_new[x] - x_old[x]));
    }
    if ( max_diff < tol) done = 1;
  } while ((iter_num < max_iter) && (!done));

  return done;
} /* Jacobi */


MATRIX_T  temp_mat;
/*********************************************************************/
void read_matrix(MATRIX_T  A_local,int n,int my_rank,int p) {
  int       i, j;
  int       n_bar;

  n_bar = n/p;
  /* Fill dummy entries in temp with zeroes */
  for (i = 0; i < n; i++)
    for (j = n; j < MAX_DIM; j++)
      temp_mat[i][j] = 0;
  if (my_rank == 0) {
    get_diag_mat(temp_mat, n, 5);
```

```
  }
  MPI_Scatter(temp_mat, n_bar*MAX_DIM, MPI_INT, A_local,
          n_bar*MAX_DIM, MPI_INT, 0, MPI_COMM_WORLD);
} /* Read_matrix */

/*********************************************************************/
void read_vector(float x_local[], int n,int my_rank, int p) {
  int   i;
  float temp[MAX_DIM];
  int   n_bar;

  n_bar = n/p;
  if (my_rank == 0) {
    get_rand_vect(temp,n);
  }
  MPI_Scatter(temp, n_bar, MPI_FLOAT, x_local, n_bar, MPI_FLOAT,
          0, MPI_COMM_WORLD);
} /* Read_vector */
```

Bibliography

[1] The High-availability Linux Project. 2000. http://www.linux-ha.org.

[2] A. ACHARYA AND B. R. BADRINATH. An efficient protocol for ordering broadcast messages in distributed systems. In *Third IEEE Symposium on Parallel and Distributed Systems*, 1991. http://paul.rutgers.edu/~acharya/publications.html.

[3] S. V. ADVE AND M. D. HILL. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*. 4(6):613–624. June 1993.

[4] D. AGRAWAL, M. CHOY, H.-V. LEONG, AND A. K. SINGH. Mixed Consistency: A Model for Parallel Programming. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing (PODC'94)*, August 1994.

[5] M. AHAMAD, P. W. HUTTO, AND R. JOHN. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, pages 274–281. Arlington, TX USA. 1991. IEEE Computer Society , Washington, DC.

[6] M. AHAMAD, G. NEIGER, P. KOHLI, J. E. BURNS, AND P. W. HUTTO. Casual Memory: Definitions, Implementation and Programming. *Distributed Computing*. 9:37–49. 1995.

[7] M. AHUJA. Flush primitives for asynchronous distributed systems. *IPL: Information Processing Letters*, 34. 1990.

[8] Y. AMIR, D. DOLEV, S. KRAMER, AND D. MALKI. Transis: A communication sub-system for high availability. In *22nd International Symposium on Fault-Tolerant Computing*, pages 76–84. IEEE, July 1992.

[9] C. AMZA, A. L. COX, S. DWARKADAS, P. KELEHER, H. LU, R. RAJAMONY, W. YU, AND W. ZWAENEPOEL. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 2(29):18–28, Feb 1996.

[10] C. AMZA, A. L. COX, S. DWARKADAS, AND W. ZWAENEPOEL. Software DSM Protocols that Adapt between Single Writer and M-ltiple Wr iter. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*. pages 261–271. February 1997.

[11] T. E. ANDERSON, D. E. CULLER, AND D. A. PATTERSON. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[12] IEEE STANDARDS ASSOCIATION. IEEE/ANSI Std 1003.1, 1996 Edition: Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language], 1996.

[13] P. AULD AND P. KEARNS. PBP: A Pipelined Broadcast Protocol for Ethernet. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems. PDCS '99*, pages 845–850, Anaheim, CA, November 1999. IASTED/ACTA Press.

[14] P. AULD AND P. KEARNS. "broadcast distributed shared memory". In *Proceedings of the ICSA 13th International Conference on Parallel and Distributed Computing Systems*, pages 225–230, ICSA, 2000.

[15] H. E. BAL, M. F. KAASHOEK, AND A. S. TANENBAUM. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):180–205, March 1992.

[16] A. BARAK AND O. LA'ADAN. Performance of the MOSIX parallel system for a cluster of PC's. *Lecture Notes in Computer Science*, 1225:624–??, 1997.

[17] R. BIANCHINI, L. I. KONTOTHANASSIS, R. PINTO, M. DE MARIA, M. ABUD, AND C. L. AMORIM. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 198–209, October 1996.

[18] K. P. BIRMAN AND T. A. JOSEPH. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[19] G. BRACHA. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[20] G. BRACHA AND S. TOUEG. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[21] T. BRECHT AND H. SANDHU. The region trap library: Handling traps on application-defined regions of memory. In *Usenix Annual Technical Conference*, pages 85–99, 1999.

[22] J. BRUCK, D. DOLEV, C. HO, R. ORNI, AND R. STRONG. PCODE: An efficient and reliable collective communication protocol for unreliable broadcast domains. In *IPPS: 9th International Parallel Processing Symposium*, pages 130–139. IEEE Computer Society Press, 1995.

[23] Rajkumar Buyya, editor. *High Performance Cluster Computing. Volume 1: Architecture and Systems*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1999.

[24] T. CAMP, P. KEARNS, AND M. AHUJA. Proof rules for flush channels. *IEEE Transactions on Software Engineering*, 19(4):366–378, April 1993.

[25] C. CAP AND V. STUMPEN. Efficient parallel computing in distributed workstation environments. *Parallel Computing*. 19:1221–1234. 1993.

[26] J. B. CARTER. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*. 29(2):219–227. September 1995.

[27] J. B. CARTER. J. K. BENNETT. AND W. ZWAENEPOEL. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. pages 152–64. Association for Computing Machinery SIGOPS. October 1991.

[28] J. B. CARTER. J. K. BENNETT. AND W. ZWAENEPOEL. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*. 13(3):205–243. August 1995.

[29] J. B. CARTER. D. KHANDEKAR. AND L. KAMB. Distributed shared memory: Where we are and where we should be headed? In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. pages 119–122. May 1995.

[30] J. CHANG AND N. F. MAXEMCHUK. Reliable Broadcast Protocols. *ACM Trans. Comp. Systems*.. 2. 3:251–273. August 1984.

[31] H. A. CHEN. Y. O. CARRASCO. AND A. W. APON. Mpi collective operations over ip multicast. In *IPDPS 2000 Workshops*. J. Rolim et Al.. editor. pages 51–60. Springer-Verlag. 2000.

[32] D. R. CHERITON. Preliminary thoughts on problem-oriented shared memory: A decentralized approach to distributed systems. *Operating Systems Review*. 19(4):26–33. October 1985.

[33] A. CHEUNG AND A. REEVES. High performance computing on a cluster of workstations. In *Proc. First Int. Symp. on High-Performance Distributed Computing*. pages 152–160. 1992.

[34] S. E. DEERING. Host Extensions for IP Multicasting. RFC 1112. Aug. 1989.

[35] D. DOLEV AND D. MALKI. The design of the Transis system. *Lecture Notes in Computer Science*. 938:83–??. 1995.

[36] L. TORVALDS ET ALIA. The Linux Kernel source tree. 1991+. http://www.kernel.org.

[37] M. FISCHER. N. LYNCH. AND M. PATERSON. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*. 32:374–382. 1985.

[38] M. J. FISCHER AND A. MICHAEL. Sacrificing serializability to attain high availability. In *ACM SIGACT-SIGMOD Symp. on Principles of Database Systems 1. Aho(ed)*. *ACM*. March 1982.

[39] MESSAGE PASSING INTERFACE FORUM. *MPI: a Message-Passing Interface Standard*. http://www.mpi-forum.org. 1995.

[40] K. GHARACHORLOO, S. ADVE, A. GUPTA, J. L. HENNESSY, AND M. D. HILL. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[41] K. GHARACHORLOO, D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA, AND J. L. HENESSY. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Int'l Symp. on Computer Architecture. ACM SIGARCH Computer Architecture News*, page 15, June 1990. Published as Proc. 17th Annual Int'l Symp. on Computer Architecture. ACM SIGARCH Computer Architecture News, volume 18, number 2.

[42] D. P. GHORMLEY, D. PETROU, S. H. RODRIGUES, A. M. VAHDAT, AND T. E. ANDERSON. GLUnix: A Global Layer Unix for a network of workstations. *Software Practice and Experience*, 28(9):929–961, July 1998.

[43] P. B. GIBBONS, M. MERRITT, AND K. GHARACHORLOO. Proving Sequential Consistency of High-Performance Shared Memories (Extended Abstract). In *Proc. of the 3rd ACM Symp. on Parallel Algorithms and Architectures (SPAA'91)*, pages 292–303, July 1991.

[44] P. GORTMAKER. Linux Ethernet-Howto. http://www.linuxdoc.org/HOWTO/Ethernet-HOWTO.html, May 1999.

[45] W. GROPP, E. LUSK, N. DOSS, AND A. SKJELLUM. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[46] M. D. HILL. Multiprocessors should support simple memory consistency protocols. *IEEE Computer*, 31(8), August 1998.

[47] W. HU, W. SHI, AND Z. TANG. Reducing system overheads in home-based software DSMs. In *Proc. of the Second Merged Symp. IPPS/SPDP 1999)*, pages 167–173, April 1999.

[48] G. HUGHES-FENCHEL. A flexible clustered approach to high availability. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 314–319, Washington - Brussels - Tokyo, June 1997. IEEE.

[49] P. W. HUTTO AND M. AHAMAD. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, pages 302–311, May 1990.

[50] L. IFTODE, C. DUBNICKI, E. FELTEN, AND K. LI. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 14–25, San Jose, California, February 3–7, 1996. IEEE Computer Society TCCA.

[51] R. JOHN AND M. AHAMAD. Casual Memory: Implementation, Programming Support and Experiences. Technical Report GIT-CC-93-10, Geogia Institute of Technology, 1993.

[52] R. JOHN AND M. AHAMAD. Evaluation of Casual Distributed Shared Memory for Data-race-free Programs. Technical Report GIT-CC-94-34. Georgia Institute of Technology, 1994.

[53] MICHAEL K. JOHNSON AND ERIK W. TROAN. *Linux Application Development.* Addison-Wesley, Reading, MA. USA, 1998.

[54] H. KARL. Bridging the gap between distributed shared memory and message passing. In *Proc. of the ACM 1998 Workshop on Java for High-Performance Network Computing.* March 1998.

[55] P. KELEHER, A. L. COX, AND W. ZWAENEPOEL. Lazy release consistency for software distributed shared memory. In *Proc. 19th Int. Symposium on Comp. Architecture*, pages 13–21, Gold Coast (Australia). May 1992.

[56] L. LAMPORT. Implementation of reliable distributed multiprocess systems. *Computer Networks: The International Journal of Distributed Informatique.* 2(2):95–114. May 1978.

[57] L. LAMPORT. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM.* 21(7):558–565. 1978.

[58] L. LAMPORT. Time. Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM.* 21(7):558–565. July 1978.

[59] L. LAMPORT. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers.* C-28(9):690–691. September 1979.

[60] D. LENOSKI, J. LAUDON, K. GHARACHORLOO, W. D. WEBER, A. GUPTA, J. HENESSY, M. HOROWITZ, AND M. S. LAM. The stanford dash multiprocessor. *IEEE Computer.* 25(3):63. March 1992.

[61] X. LEROY. Linuxthreads - POSIX 1003.1c kernel threads for Linux. Software Library. 1997. http://pauillac.inria.fr/~xleroy/linuxthreads.

[62] K. LI AND P. HUDAK. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359. 1986.

[63] R. J. LIPTON AND J. S. SANDBERG. PRAM: A scalable shared memory. Technical Report CS-TR-180-88. Princeton University, September 1988.

[64] J. LONCARIC. Linux 2.0.36 TCP Performance Fix for Short Messages. 1999. http://www.icase.edu/coral/LinuxTCP.html.

[65] R. M. METCALF AND D. R. BOGGS. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[66] L. E. MOSER, P. M. MELLIAR-SMITH, D. A. AGARWAL, R. K. BUDHIA, AND C. A. LINGLEY-PAPADOPOULOS. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[67] B. NITZBERG AND V. LO. Distributed shared memory: A survey of issues and algorithms. *Computer*, pages 52–60. August 1991.

[68] M. OGUCHI, H. AIDA, AND T. SAITO. A proposal for a DSM architecture suitable for a widely distributed environment and its evaluation. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 32–39. August 1995.

[69] P. PACHECO. *Parallel programming with MPI.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1997.

[70] C. PARTRIDGE AND R. HINDEN. Version 2 of the Reliable Datagram Protocol (RDP). RFC 1151. April 1990. 4 Pages.

[71] L. L. PETERSON, N. C. BUCHHOLZ, AND R. D. SCHLICHTING. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[72] U. RAMACHANDRAN AND M. Y. A. KHALIDI. An implementation of distributed shared memory. *Software, Practice and Experience*, 21(5):443–464. [5] 1991.

[73] M. RAYNAL AND A. SCHIPER. From Casual Consistency to Sequential Consistency in Shared Memory Systems. Technical Report 926. IRISA. France. May 1995.

[74] G. RICART AND A. AGRAWAL. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17. Jan 1981.

[75] D. J. SCALES AND K. GHARACHORLOO. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[76] D. J. SCALES, K. GHARACHORLOO, AND C. A. THEKKATH. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Sha red Memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, October 1996.

[77] F. B. SCHNEIDER. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, April 1982.

[78] F. B. SCHNEIDER. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319. December 1990.

[79] T. SEIDMANN. Multicast-based runtime system for highly efficient causally consistent software-only DSM. *Lecture Notes in Computer Science*, 1586:547–??, 1999.

[80] J. P. SINGH, W. WEBER, AND A. GUPTA. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1995.

[81] IEEE COMPUTER SOCIETY. IEEE CS Task Force on Cluster Computing, 2000. http://www.ieeetfcc.org/.

[82] E. SPEIGHT AND J. K. BENNETT. Brazos: A Third Generation DSM System. In *Proc. of the USENIX Windows NT Workshop*. August 1997.

[83] T. STERLING, D. BECKER, AND MORE. The Beowulf Project at CESDIS. http://beowulf.gsfc.nasa.gov/.

[84] T. STERLING, D. SAVARESE, D. J. BECKER, J. E. DORBAND, U. A. RANAWAKE, AND C. V. PACKER. BEOWULF : A parallel workstation for scientific computation. In *International Conference on Parallel Processing, Vol.1: Architecture*, pages 11–14. Boca Raton, USA. August 1995. CRC Press.

[85] W. R. STEVENS. *TCP/IP Illustrated- The Protocols*. Addison-Wesley, Reading, MA, USA, 1994.

[86] W. R. STEVENS. *UNIX network programming: Networking APIs: sockets and XTI*. volume 1. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, second edition, 1998.

[87] M. STUMM AND S. ZHOU. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, [5] 1990.

[88] M. SWANSON, L. STROLLER, AND J. B. CARTER. Making distributed shared memory simple, yet efficient. In *Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'98)*, pages 2–13. March 1998.

[89] A. S. TANENBAUM, M. F. KAASHOEK, AND H. E. BAL. Using broadcasting to implement distributed shared memory efficiently. In *Readings in Distributed Computing Systems*, T. L. Casavant and M. Singhal, editors, pages 387–408. IEEE Computer Society Press, 1994.

[90] ANDREW S. TANENBAUM. *Computer Networks*. Prentice Hall, 2. edition, 1989.

[91] S. TOUEG, K. J. PERRY, AND T. K. SRIKANTH. Fast distributed agreement. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, Ray Strong, editor, pages 87–101, Minaki, ON, Canada, August 1985. ACM Press.

[92] E. UPFAL AND A. WIGDERSON. How to share memory in a distributed system. *Journal of the Association for Computing Machinery*, 34(1):116–127, [1] 1987.

[93] D. WALKER. Long-range n-body code. Web Page, 1995. http://www.epm.ornl.gov/~walker/OLD_ORNL_WEB_PAGE/mpi/examples/nbody.html.

[94] B. WHETTEN, T. MONTGOMERY, AND S. KAPLAN. A High Performance Totally Ordered Multicast Protocol. *Lecture Notes in Computer Science*, 938:33–55, 1995.

[95] G. WRIGHT AND W. R. STEVENS. *TCP/IP Illustrated- The Implementation*. Addison-Wesley, Reading, MA, USA, 1995.

[96] M. J. ZEKAUSKAS, W. A. SAWDON, AND B. N. BERSHAD. Software write detection for a distributed shared memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 87–100, November 1994.

VITA

Born in Northampton, Massachusetts. Philip was raised in the Research Triangle Park area of N.C. After resettling in Connecticut, he did his undergraduate work at Hunter College in New York City. He started graduate school in 1993 and completed a Master's degree in 1995. Married in June of 2000, he now lives in Newton, Mass with his wife Catherine. He is currently doing kernel engineering and product development for Egenera Inc.